PERFORMANCE IMPROVEMENT OF MULTITHREADED JAVA APPLICATIONS EXECUTION ON MULTIPROCESSOR SYSTEMS

by

Jordi Guitart Fernández

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)
July 2005

Dr.	
President	

Dr.	
Secretari	

Dr.	
Vocal	

Dr.	
Vocal	

Dr.	
Vocal	

Data de la defensa pública	
Qualificació	

ABSTRACT

The design of the Java language, which includes important aspects such as its portability and architecture neutrality, its multithreading facilities, its familiarity (due to its resemblance with C/C++), its robustness, its security capabilities and its distributed nature, makes it a potentially interesting language to be used in parallel environments such as high performance computing (HPC) environments, where applications can benefit from the Java multithreading support for performing parallel calculations, or e-business environments, where multithreaded Java application servers (i.e. following the J2EE specification) can take profit of Java multithreading facilities to handle concurrently a large number of requests.

However, the use of Java for parallel programming has to face a number of problems that can easily offset the gain due to parallel execution. The first problem is the large overhead incurred by the threading support available in the JVM when threads are used to execute fine-grained work, when a large number of threads are created to support the execution of the application or when threads closely interact through synchronization mechanisms. The second problem is the performance degradation occurred when these multithreaded applications are executed in multiprogrammed parallel systems. The main issue that causes these problems is the lack of communication between the execution environment and the applications, which can cause these applications to make an uncoordinated use of the available resources.

This thesis contributes with the definition of an environment to analyze and understand the behavior of multithreaded Java applications. The main contribution of this environment is that all levels in the execution (application, application server, JVM and operating system) are correlated. This is very important to understand how this kind of applications behaves when executed on environments that include servers and virtual machines, because the origin of performance problems can reside in any of these levels or in their interaction.

In addition, and based on the understanding gathered using the proposed analysis environment, this thesis contributes with scheduling mechanisms and policies oriented towards the efficient execution of multithreaded Java applications on multiprocessor systems considering the interactions and coordination between scheduling mechanisms and policies at the different levels involved in the execution. The basis idea consists of allowing the cooperation between the applications and the execution environment in the resource management by establishing a bi-directional communication path between the applications and the underlying system. On one side, the applications request to the execution environment the amount of resources they need. On the other side, the execution environment can be requested at any time by the applications to inform them about their resource assignments.

This thesis proposes that applications use the information provided by the execution environment to adapt their behavior to the amount of resources allocated to them (self-adaptive applications). This adaptation is accomplished in this thesis for HPC environments through the malleability of the applications, and for e-business environments with an overload control approach that performs admission control based on SSL connections differentiation for preventing throughput degradation and maintaining Quality of Service (QoS).

The evaluation results demonstrate that providing resources dynamically to self-adaptive applications on demand improves the performance of multithreaded Java applications as in HPC environments as in e-business environments. While having self-adaptive applications avoids performance degradation, dynamic provision of resources allows meeting the requirements of the applications on demand and adapting to their changing resource needs. In this way, better resource utilization is achieved because the resources not used by some application may be distributed among other applications.

ACKNOWLEDGEMENTS

It is a pleasure to thank all those people who has contributed to make this thesis possible. I would like to especially acknowledge my Ph.D. advisors, Dr. Jordi Torres and Dr. Eduard Ayguadé, for their guidance and support over the last years that have made possible to accomplish this work. I am indebted with all the members of the eDragon Research Group, especially David Carrera and Vicenç Beltran, for their useful comments and suggestions and our valuable discussions. Special thanks also to Dr. Jesús Labarta, Dr. Xavier Martorell, Dr. Mark Bull, José Oliver, Alex Duran and all the people that have collaborated in some way in this work. Finally, I would like to thank all my colleagues in the Computer Architecture Department for these years and my family for their support and patience.

The research described in this dissertation has been supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIC98-511, TIC2001-0995-C02-01, TIN2004-07739-C02-01 and by the Generalitat de Catalunya under grant 2001FI 00694 UPC APTIND. I acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying the computing resources for some of the experiments in this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 Introduction

Over the last years, Java has consolidated as an interesting language for the network programming community. This has largely occurred as a direct consequence of the design of the Java language. This design includes, among others, important aspects such as the portability and architecture neutrality of Java code, and its multithreading facilities. The latter is achieved through built-in support for threads in the language definition. The Java library provides the Thread class definition, and Java runtimes provide support for thread, mutual exclusion and synchronization primitives. These characteristics, besides others like its familiarity (due to its resemblance with C/C++), its robustness, its security capabilities and its distributed nature also make it a potentially interesting language to be used in parallel environments.

For instance, the Java language could be used in high performance computing (HPC) environments, where applications can benefit from the Java multithreading support for performing parallel calculations. In the same way, Internet applications programmers also use Java when developing these applications. Thus, it is common to find Internet servers following the Java 2 Platform Enterprise Edition [132] (J2EE) specification (i.e. written in Java), as for instance Tomcat [84] and Websphere [146], hosting current web sites. Typically, these servers are multithreaded Java applications in charge of serving clients requesting for web content, where each client connection is assigned to a thread that is the responsible of attending the received requests in this connection. Thus, the servers can take profit of Java multithreading facilities to handle concurrently a large number of requests.

However, although recent results show how the performance gap between Java and other traditional languages is being reduced [24], and some language extensions [23] and runtime support have been proposed [111] to ease the specification of Java parallel applications and make threaded execution more efficient, the use of Java for parallel programming has still to face a number of problems that can easily offset the gain due to parallel execution. The first problem is the large overhead incurred by the threading support available in the JVM when threads are used to execute fine-grained work, when a large number of threads are created to support the execution of the application or when threads closely interact through synchronization mechanisms. The second problem is the performance degradation occurred when these multithreaded applications are executed in multiprogrammed parallel systems. The main issue that causes these problems is the lack of communication between the execution environment and the applications, which can cause these applications to make an uncoordinated use of the available resources.

This thesis contributes with the definition of an environment to analyze and understand the behavior of multithreaded Java applications. The main contribution of this environment is that all levels in the execution (application, application server, JVM and operating system) are correlated. This is very important to understand how this kind of applications behaves when executed on environments that include servers and virtual machines.

In addition, and based on the understanding gathered using the proposed analysis environment, this thesis proposes research on scheduling mechanisms and policies oriented towards the efficient execution of multithreaded Java applications on multiprocessor systems considering the interactions and coordination between scheduling mechanisms and policies at different levels: application, application server, JVM, threads library and operating system.

In order to achieve these main objectives, the thesis is divided in the following work areas.

- Analysis and Visualization of Multithreaded Java Applications
- Self-Adaptive Multithreaded Java Applications
- Resource Provisioning for Multithreaded Java Applications

1.2 Contributions

1.2.1 Analysis and Visualization of Multithreaded Java Applications

Previous experience on parallel applications has demonstrated that tuning this kind of applications for performance is mostly responsibility of (experienced) programmers [93]. Therefore, the performance analysis of multithreaded Java applications can be a complex work due to this inherent difficulty of analyzing parallel applications as well as the extra complexity added by the presence of the Java Virtual Machine. In this scenario, performance analysis and visualization tools that provide detailed information of multithreaded Java applications behavior are necessary in order to help users in the process of tuning the applications on the target parallel systems and JVM.

In the same way, the increasing load that the applications currently developed for Internet must support, demands new performance requirements to the Java application servers that host them. To achieve these performance requirements, fine-grain tuning of these servers is needed, but this tuning can be a hard work due to the large complexity of these environments (including the application server, distributed clients, a database server, etc.). Tuning Java application servers for performance requires also of tools that allow an in-depth analysis of application server behavior and its interaction with the other system elements. These tools must consider all levels involved in the execution of web applications (operating system, JVM, application server and application) if they want to provide significant performance information to the administrators (the origin of performance problems can reside in any of these levels or in their interaction).

Although a number of tools have been developed to monitor and analyze the performance of multithreaded Java applications (see Section 6.1), none of them allow a fine-grain analysis of the applications behavior considering all levels involved in the application execution. The main contribution in the “Analysis and Visualization of Multithreaded Java Applications” work area of this thesis is the proposal of a performance analysis framework to perform a complete analysis of the Java applications behavior. This framework provides to the user detailed and correlated information about all levels involved in the application execution, giving him the chance to construct his own metrics, oriented to the kind of analysis he wants to perform.

The performance analysis framework consists of two tools: an instrumentation tool, called JIS (Java Instrumentation Suite), and an analysis and visualization tool, called Paraver [116]. When instrumenting a given application, JIS generates a trace in which the information collected from all levels has been correlated and merged. The trace reflects the activity of each thread in the application recorded in the form of a set of predefined state transitions (that are representative of the parallel execution) and the occurrence of some predefined events. Later, the trace can be visualized and analyzed with Paraver (qualitatively and quantitatively) to identify the performance bottlenecks of the application.

The instrumentation tool (JIS) is responsible of collecting detailed information from all levels involved in the execution of Java applications. From the system level, information about threads state and system calls (I/O, sockets, memory management and thread management) can be obtained. Several implementations are proposed depending on the underlying platform. A dynamic interposition mechanism that obtains information about the supporting threads layer (i.e. Pthreads library [121]) without recompilation has been implemented for the SGI Irix platform. In the same way, a device driver that gets information from a patched Linux kernel has been developed for the Linux platform. JIS uses the Java Virtual Machine Profiler Interface [143] (JVMPI) to obtain information from the JVM level. JVMPI is a common interface designed to introduce hooks inside the JVM code in order to be notified about some predefined Java events. At this level of analysis, the user can obtain information about several Java abstractions like classes, objects, methods, threads and monitors, but JIS only obtains at this level the name of the Java threads and the operations performed on the different Java Monitors, due to the large overhead produced when using JVMPI. Information relative to services (i.e. Java Servlets [136] and Enterprise Java Beans [131] (EJB)), requests, connections or transactions can be obtained from the application server level. Moreover, some extra information can be added to the final trace file by generating user events from the application code. Information at these levels can be inserted by hard-coding hooks to a Java Native Interface [134] (JNI) on the server or the application source or by introducing them dynamically using Aspect programming techniques [60] without source code recompilation.

As a special case of instrumentation at the application level, support for JOMP applications [23] is included in JIS. JOMP includes OpenMP-like extensions to specify parallelism in Java applications using a shared-memory programming paradigm. This instrumentation approach provides a detailed analysis of the parallel behavior at the JOMP programming model level. At this level, the user is faced with parallel, work-sharing and synchronization constructs. The JOMP compiler has been modified to inject JNI calls to the instrumentation library during the code generation phase at specific points in the source code.

1.2.2 Self-Adaptive Multithreaded Java Applications

Multithreaded Java applications can be used in HPC environments, where applications can benefit from the Java multithreading support for performing parallel calculations, as well as in e-business environments, where Java application servers can take profit of Java multithreading facilities to handle concurrently a large number of requests.

However, the use of Java for HPC faces a number of problems that are currently subject of research. One of them is the performance degradation when multithreaded applications are executed in a multiprogrammed environment. The main issue that leads to this degradation is the lack of communication between the execution environment and the applications, which can cause these applications to make a naive use of threads, degrading their performance. In these situations, it is desirable that the execution environment provides information to the applications about their allocated resources, thus allowing the applications to adapt their behavior to the amount of resources offered by the execution environment by generating only the amount of parallelism that can be executed with the assigned processors. This capability of applications is known as malleability [53]. Therefore, improving the performance of multithreaded Java applications in HPC environments can be accomplished by designing and implementing malleable applications (i.e. self-adaptive applications).

Achieving good performance when using Java in e-business environments is a harder problem due to the high complexity of these environments. First, the workload of Internet sites is known to vary dynamically over multiple time scales, often in an unpredictable fashion, including flash crowds. This fact and the increasing load that Internet sites must support increase the performance demand on Java application servers that host the sites that must face situations with a large number of concurrent clients. Therefore, the scalability of these application servers has become a crucial issue in order to support the maximum number of concurrent clients in these situations.

Moreover, not all the web requests require the same computing capacity from the server. For example, requests for static web content (i.e. HTML files and images) are mainly I/O intensive. Requests for dynamic web content (i.e. servlets and EJB) increase the computational demand on server, but often other resources (e.g. the database) become the bottleneck for performance. On the other side, in e-business applications, which are based on dynamic web content, all information that is confidential or has market value must be carefully protected when transmitted over the open Internet. These security capabilities between network nodes over the Internet are traditionally provided using HTTPS [125]. With HTTPS, which is based on using HTTP over SSL (Secure Socket Layer [56]), mutual authentication of both the sender and receiver of messages is performed to ensure message confidentiality. Although providing these security capabilities does not introduce a new degree of complexity in web applications structure, it increases the computation time necessary to serve a connection remarkably, due to the use of cryptographic techniques, becoming a CPU-intensive workload.

Facing situations with a large number of concurrent clients and/or with a workload that demands high computational power (as for instance secure workloads) can lead a server to overload (i.e. the volume of requests for content at a site temporarily exceeds the capacity for serving them and renders the site unusable). During overload conditions, the response times may grow to unacceptable levels, and exhaustion of resources may cause the server to behave erratically or even crash causing denial of services. In e-commerce applications, which are heavily based on the use of security, such server behavior could translate to sizable revenue losses. For instance, [150] estimates that between 10 and 25% of e-commerce transactions are aborted because of slow response times, which translates to about 1.9 billion dollars in lost revenue. Therefore, overload prevention is a critical issue if good performance on Java application servers in e-business environments wants to be achieved. Overload prevention tries to have a system that remains operational in the presence of overload even when the incoming request rate is several times greater than system capacity, and at the same time is able to serve the maximum the number of requests during such overload, maintaining response times (i.e. Quality of Service (QoS)) within acceptable levels.

Additionally, in many web sites, especially in e-commerce, most of the applications are session-based. A session contains temporally and logically related request sequences from the same client. Session integrity is a critical metric in e-commerce. For an online retailer, the higher the number of sessions completed the higher the amount of revenue that is likely to be generated. The same statement cannot be made about the individual request completions. Sessions that are broken or delayed at some critical stages, like checkout and shipping, could mean loss of revenue to the web site. Sessions have distinguishable features from individual requests that complicate the overload control. For example, admission control on per request basis may lead to a large number of broken or incomplete sessions when the system is overloaded.

Application servers overload can be prevented by designing mechanisms that allow the servers to adapt their behavior to the available resources (i.e. becoming self-adaptive applications) limiting the number of accepted requests to those that can be served without degrading their QoS while prioritizing important requests. However, the design of a successful overload prevention strategy must be preceded by a complete characterization of the application server scalability. This characterization allows determining which factors are the bottlenecks for application server performance that must be considered in the overload prevention strategy.

Nevertheless, characterizing application server scalability is something more complex than measuring the application server performance with different number of clients and determining the load that overloads the server. A complete characterization must also supply the causes of this overload, giving to the server administrator the chance and the information to improve the server scalability by avoiding its overload. For this reason, this characterization requires of 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.). As described in Section 1.2.1, these tools must support and consider all the levels involved in the execution of web applications if they want to provide meaningful 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: the scalability relative to the resources. The analysis of the application server behavior will assist with hints to answer the question about how would affect to the application server scalability the addition of more resources. If the analysis reveals that some resource is being a bottleneck for the application server performance, this encourages the addition of new resources of this type in order to improve server scalability. On the other side, if a resource that is not being a bottleneck for the application server performance is upgraded, the added resources are wasted because the scalability is not improved and the causes of server performance degradation remain unresolved.

The first contribution of this thesis in the “Self-Adaptive Multithreaded Java Applications” work area is a complete characterization of the scalability of Java application servers when running secure dynamic web applications divided in two parts. The first part consists of measuring Tomcat vertical scalability (i.e. adding more processors) when using SSL determining the impact of adding more processors on server overload. The second part involves a detailed analysis of the server behavior using the performance analysis framework mentioned in Section 1.2.1, in order to determine the causes of the server overload when running with different number of processors.

The conclusions derived from this analysis demonstrate the convenience of incorporating to the application server (and give hints for its implementation) an overload control mechanism that is the second contribution of this thesis in the “Self-Adaptive Multithreaded Java Applications” work area. The overload control mechanism is based on SSL connections differentiation and admission control. SSL connections differentiation is accomplished by proposing a possible extension of the Java Secure Sockets Extension [135] (JSSE) package to distinguish SSL connections depending on if the connection will reuse an existing SSL connection on the server or not. This differentiation can be very useful in order to design intelligent overload control policies on server, given the big difference existing on the computational demand of new SSL connections versus resumed SSL connections. Based on this SSL connections differentiation, a session-based adaptive admission control mechanism for the Tomcat application server is implemented. This mechanism allows the server to avoid throughput degradation and response time increments occurred on server saturation. The server differentiates full SSL connections from resumed SSL connections limiting the acceptation of full SSL connections to the maximum number acceptable with the available resources without overloading, while accepting all the resumed SSL connections. Moreover, the admission control mechanism maximizes the number of sessions completed successfully, allowing to e-commerce sites based on SSL to increase the number of transactions completed, thus generating higher benefit.

1.2.3 Resource Provisioning for Multithreaded Java Applications

In the way towards achieving good performance when running multithreaded Java applications, either in HPC environments or in e-business environments, this thesis demonstrates that having self-adaptive multithreaded Java applications can be very useful to achieve this objective.

However, the maximum effectiveness for preventing applications performance degradation in parallel environments is obtained when fitting the self-adaptation of the applications to the available resources within a global strategy in which the execution environment and the applications cooperate to manage the resources efficiently.

For example, besides of having self-adaptive Java applications in HPC environments, performance degradation of multithreaded Java applications in these environments can only be avoided if overcoming the following limitations. First, the Java runtime environment does not allow applications to have control on the number of kernel threads where Java threads map and to suggest about the scheduling of these kernel threads. Second, the Java runtime environment does not inform the applications about the dynamic status of the underlying system so that the applications cannot adapt their execution to these characteristics. Finally, the large number of migrations of the processes allocated to an application occurred, due to scheduling polices that do not consider multithreaded Java applications as an allocation unit.

The same applies to Java application servers in e-business environments. In this case, although the admission control mechanisms used to implement self-adaptive applications in this scenario can maintain the quality of service of admitted requests even during overloads, a significant fraction of the requests may be turned away during extreme overloads. In such a scenario, an increase in the effective server capacity is necessary to reduce the request drop rate. In fact, although several techniques have been proposed to face with overload, such as admission control, request scheduling, service differentiation, service degradation or resource management, last work in this area has demonstrated that the most effective way to handle overload considers a combination of these techniques [140].

For these reasons, this thesis contributes in the "Resource Provisioning for Multithreaded Java Applications" work area with the proposal of mechanisms to allow the cooperation between the applications and the execution environment in order to improve the performance by managing resources efficiently in the framework of Java applications, including the modifications that are required in the Java execution environment to allow this cooperation. The cooperation is implemented by establishing a bi-directional communication path between the applications and the underlying system. On one side, the applications request to the execution environment the number of processors they need. On the other side, the execution environment can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications, which are self-adaptive, can adapt their behavior to the amount of resources allocated to them.

In order to accomplish this resource provisioning strategy in HPC environments, this thesis shows that the services supplied by the Java native underlying threads library, in particular the services to inform the library about the concurrency level of the application, are not enough to support the cooperation between the applications and the execution environment, because this uni-directional communication does not allow the application to adapt its execution to the available resources. In order to address the problem, the thesis proposes to execute the self-adaptive multithreaded Java applications on top of JNE (Java Nanos Environment built around the Nano-threads environment [101]). JNE is a research platform that provides mechanisms to establish a bi-directional communication path between the Java applications and the execution environment, thus allowing applications to collaborate in the thread management. Running with JNE, the applications can inform to the execution environment about their processor requirements, as well as, JNE allows to the execution environment to answer to applications with the number of processors assigned to them at any moment. The JNE scheduler is responsible for the distribution of processors to applications and decides which processors are assigned to each application taking into account data affinity issues (i.e. helping the application to exploit data locality whenever possible). As the applications are malleable (i.e. self-adaptive), they can adapt their behavior to the amount of resources offered by the execution environment. The work in this area includes the adaptation of JOMP applications in order to cooperate with the execution environment. The implementation of the JOMP compiler and supporting runtime library has been modified to implement the communication between the JOMP application and the JNE. The generated code will adapt its parallelism level depending on the available processors at a given time.

The global resource provisioning strategy is accomplished in e-business environments using an overload control approach for self-adaptive Java application servers running secure e-commerce applications that brings together admission control based on SSL connections differentiation and dynamic provisioning of platform resources in order to adapt to changing workloads avoiding the QoS degradation. Dynamic provisioning enables additional resources to be allocated to an application on demand to handle workload increases, while the admission control mechanisms maintain the QoS of admitted requests by turning away excess requests and preferentially serving preferred clients (to maximize the generated revenue) while additional resources are being provisioned.

The overload control approach is based on a global resource manager responsible of distributing periodically the available resources (i.e. processors) among web applications in a hosting platform applying a given policy (which can consider e-business indicators). This resource manager and the applications cooperate to manage the resources using a bi-directional communication. On one side, the applications request to the resource manager the number of processors needed to handle their incoming load avoiding the QoS degradation. On the other side, the resource manager can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications, which are self-adaptive, apply the admission control mechanism presented in Section 1.2.2 to adapt their incoming workload to the assigned capacity by limiting the number of admitted requests accepting only those that can be served with the allocated processors without degrading their QoS.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 describes multithreaded Java applications, which are the focus of this work, distinguishing the use of this kind of applications in HPC environments as well as e-business environments. Chapter 3 presents the performance analysis framework that allows the analysis and the visualization of multithreaded Java applications. Chapter 4 introduces self-adaptive applications in order to improve the performance of multithreaded Java applications. Chapter 5 presents the mechanisms that allow to the applications to cooperate with the execution environment in the resource management. Chapter 6 describes the related work and finally, Chapter 7 presents the conclusions and the future work of this thesis.

CHAPTER 2
MULTITHREADED JAVA APPLICATIONS

2.1 Introduction

The work performed in this thesis targets multithreaded Java applications. In the last years, these applications have been successfully introduced in high performance computing (HPC) environments, where Java applications can benefit from the Java multithreading support for performing parallel calculations. Moreover, they have also achieved a great diffusion in e-business environments based on Java application servers that can take profit of Java multithreading facilities to handle concurrently a large number of requests. This popularity of Java applications has occurred as a consequence of some Java language characteristics, which can be summarized as follows:

- ✓ **Java is familiar and simple**. Java builds on the familiar and useful features of C++ while removing its complex, dangerous, and superfluous elements. The result is a language that is safer, simpler, and easier to use.
- ✓ **Java is platform independent**. A Java program can be executed in any platform without recompilation. This portability is accomplished by offering a binary code (called 'bytecode') that is interpreted by a virtual machine.
- ✓ **Java is object-oriented**. Java provides all the luxuries of object-oriented programming: class hierarchy, inheritance, encapsulation, and polymorphism- in a context that is truly useful and efficient. Object-oriented software is simple and favors software reuse.
- ✓ **Java is safe**. Java provides security on several different levels. First, the language was designed to make it extremely difficult to execute damaging code. The elimination of pointers is a big step in this regard. Another level of security is the bytecode verifier. Before a Java program is run, the verifier checks each bytecode to make sure that nothing suspicious is going on. In

addition to these measures, Java implements a security model, known as the "sandbox" model, that provides a very restricted environment in which to run untrusted code obtained from the open network. In the sandbox model, local code is trusted to have full access to vital system resources, such as the file system, but downloaded remote code (an applet) is not trusted and can access only the limited resources provided inside the sandbox. A security manager is responsible of determining which resource accesses are allowed. Finally, the Java library provides classes that allow accessing and developing cryptographic functionality (including digital signature algorithms, message digest algorithms, key-generation algorithms and certificates management).

- ✓ **Java is extensible**. Java allows the definition of native methods written in other languages (such as C, C++, and assembly) to handle those situations when an application cannot be written entirely in the Java programming language.
- ✓ **Java is 'garbage collected'**. Java automatically frees memory occupied by unreferenced objects.
- ✓ **Java supports parallel computing**. The Java library provides the Thread class definition, and Java runtime provides support for thread and thread synchronization primitives (e.g. monitors).
- ✓ **Java supports distributed computing.** The Java library provides classes supporting the communication of applications over the network. These classes implement sockets (connection-oriented communications using TCP protocol), secure sockets (sockets that transmit encrypted information), datagrams (not connection-oriented communications using UDP protocol), URLs (references or addresses to resources on the Internet) and Remote Methods Invocation (RMI).
- ✓ **Java technology is organized in subject areas.** The Java technology distinguishes several conceptual areas, providing different solutions for each of these areas. For example, the Java 2 Platform Standard Edition [133] (J2SE) provides a complete environment for applications development on desktops and servers and for deployment in embedded environments, serving also as the foundation for the other areas. This solution is used for developing Java applications in HPC environments. On the other side, Java technology offers

also the Java 2 Platform Enterprise Edition [132] (J2EE), which defines the standard for developing component-based multi-tier enterprise applications. This solution is used for developing Java applications in e-business environments.

2.2 Multithreaded Java Applications in HPC Environments

2.2.1 Introduction

Although the Java language is a potentially interesting language for parallel scientific computing, it has to face a set of problems that must be resolved to consolidate Java as a valid alternative to the traditional languages in HPC environments. The first one is the large overhead incurred by the threading support available in the JVM. Recent works [111] offer runtime support to make threaded execution more efficient by reducing the number of creations of Java threads. The second problem is the performance degradation occurred when multithreaded applications are executed in multiprogrammed parallel systems, which is covered in this thesis. Other drawbacks include the lack of support for complex numbers and multi-dimensional arrays, which has been also addressed in previous works [26] by proposing extensions to the Java language. Finally, the lack of suitable standards to ease parallel programming in Java is also a concern when targeting high performance computing, because it implies the explicit management of parallelism and synchronization. This fact has motivated several proposals to support the specification of parallelism in Java, which are discussed in next section.

2.2.2 Java Extensions for High Performance Computing

Most of the current proposals to support the specification of parallel algorithms in Java reflect the alternatives that have been proposed for other languages such as Fortran or C. For instance, there have been proposals to implement common message-passing standards, such as PVM [58] or MPI [103], by means of Java classes [54][87]. Other proposals [26] try to make Java a data-parallel language similar to HPF, in which parallelism could be expressed in a more natural way. The extensions allow the definition of data-parallel operations, non-rectangular or multi-dimensional arrays or to allow some kind of data locality. The OpenMP standard [113] for Fortran and C/C++ has led to the proposal of a similar paradigm in the scope of Java (JOMP

[23]) and the automatic restructuring of Java programs for parallelism exploitation based either on code annotations or compiler-driven analysis [20][21]. The implementation of these extensions is done through runtime libraries and compiler transformations in order to avoid the overhead introduced by the intensive creation of Java Threads [23][111].

Some experiments in this thesis use JOMP applications as the benchmark to evaluate the proposed mechanisms, as a particular case of multithreaded Java applications in HPC environments. For this reason, next section presents an in-depth description of JOMP applications implementation.

2.2.3 JOMP Programming Model

The JOMP programming model [23], proposed by the Edinburgh Parallel Computing Center [52], consists in a collection of compiler directives, library routines and environment variables based on OpenMP [113] to specify shared-memory parallelism in Java.

The JOMP specification for Java includes parallel, work-sharing and synchronization constructs. The `parallel` directive is used to specify the creation of a team of threads that will concurrently execute the code. Work-sharing directives are provided to allow the distribution of work among the threads in a team: `for` directive to distribute iterations in a parallel loop, `sections` directive to parcel out a sequence of statements and `master` and `single` directives to specify the execution by a single thread in the team. Parallel and work-sharing constructs also allow redefining the scope of certain variables in order to be `shared`, `private`, `firstprivate`, `lastprivate` or `reduction`. Synchronization directives provide the mechanisms to synchronize the execution of the threads in the team: `barrier` and `critical` regions.

2.2.3.1 JOMP compiler

The JOMP compiler is a Java-to-Java translator that interprets JOMP directives and generates parallel code for the JOMP supporting runtime.

A description of JOMP compiler implementation is presented below. Additional implementation details about the API and implementation can be found in elsewhere [25][88]. Currently, a few parts of the specification have yet to be implemented, such as nested parallelism and array reductions.

2.2.3.1.1 Basic structure

The JOMP Compiler is built around a Java 1.1 parser provided as an example with the JavaCC [104] utility. JavaCC comes supplied with a grammar to parse a Java 1.1 program into a tree, and an `UnparseVisitor` class, which unparses the tree to produce code. The bulk of the JOMP compiler is implemented in the `OMPVisitor` class, which extends the `UnparseVisitor` class, overriding various methods that unparse particular non-terminals. These overriding methods output modified code, which includes calls to the supporting runtime library to implement appropriate parallelism. Because JavaCC is itself written in Java, and outputs Java source, the JOMP system is fully portable, requiring only a JVM installation in order to run it.

2.2.3.1.2 The `parallel` directive

Upon encountering a `parallel` directive within a method, the compiler creates a new class. The new class has a single method `go()`, which takes a parameter indicating an absolute thread identifier. For each variable declared to be `private`, `firstprivate` or `reduction`, the `go()` method declares a local variable with the same name and type signature. The local `firstprivate` variables are initialized from the corresponding field in the containing class, while the local `private` variables have the default initialization. The local `reduction` variables are initialized with the appropriate default value for the reduction operator. Private objects are allocated using the default constructor. The main body of the `go()` method contains the code to be executed in parallel.

In place of the parallel construct itself, code is inserted to declare a new instance of the compiler-created class, and to initialize the fields within it from the appropriate variables. The `OMP.doParallel()` method is used to execute the `go()` method of the inner class in parallel. Finally, any necessary values are copied from class fields back into local variables.

A simple example illustrating the code transformation made by JOMP compiler is shown in Figure 2.1. Figure 2.1.a shows the source code of a simple program with a `parallel` directive. This means that all the threads in the team will concurrently execute the code encapsulated by this directive. The directive has also a `private` clause for the `myid` variable, informing the compiler to allocate a private copy of this variable for each thread (usually in the stack of the encapsulating method).

```
public class Hello {
  public static void main (String argv[]) {
    int myid;
    //omp parallel private (myid)
    {
      myid = OMP.getThreadNum();
      System.out.println("Hello from" + myid);
    }
  }
}
```

(a) original code

```
public class Hello {
  public static void main (String argv[]) {
    int myid;
    // OMP PARALLEL BLOCK BEGINS
    {
      __omp_Class0 __omp_Object0 = new __omp_Class0();
      __omp_Object0.argv = argv;
      try {
        jomp.runtime.OMP.doParallel(__omp_Object0);
      } catch(Throwable __omp_exception) {
        jomp.runtime.OMP.errorMessage();
      }
      argv = __omp_Object0.argv;
    }
    // OMP PARALLEL BLOCK ENDS
  }
}

// OMP PARALLEL REGION INNER CLASS DEFINITION BEGINS
private static class __omp_Class0 extends jomp.runtime.BusyTask {
  String [] argv;
  public void go(int __omp_me) throws Throwable {
    int myid;
    // OMP USER CODE BEGINS
    {
      myid = OMP.getThreadNum();
      System.out.println("Hello from" + myid);
    }
    // OMP USER CODE ENDS
  }
}
// OMP PARALLEL REGION INNER CLASS DEFINITION ENDS
```

(b) transformed code

Figure 2.1. Example of code transformation made by the JOMP compiler: `parallel` *directive*

Like is shown in Figure 2.1.b, on encountering a `parallel` directive, the compiler creates a new class that extends the `BusyTask` class. The new class has a `go()` method, containing the code inside the parallel region, and declarations of private variables like `myid`. The new class contains also data members corresponding to reduction and shared variables like `argv`. A new instance of the class is created, and passed to the JOMP runtime library calling the `doParallel()` method, which causes the `go()` method to be executed on each thread in the team.

2.2.3.1.3 Work-sharing directives

Upon encountering a `for`, `sections` or `single` directive, a new block is created. For each variable declared to be `private`, `firstprivate`, `lastprivate` or `reduction`, a local variable is declared and initialized if necessary. These newly created variables are used to communicate the values of variables to the enclosing block. In the case of the `for` and `sections` directives, it is also necessary to declare a boolean variable to hold information about whether the current thread is the one performing the sequentially last iteration of the loop or the sequentially last section.

Inside the newly allocated block, a second block is created. For each variable declared to be `private`, `firstprivate`, `lastprivate`, or `reduction`, a new variable with the same name is declared. Variables declared as `reduction` are initialized with the appropriate value. `private` and `lastprivate` variables are initialized by calling the default constructor in the case of class type variables, or left uninitialized in the case of primitive or array type variables. `firstprivate` variables are initialized with the appropriate value from the original variable. A `clone()` method is called to initialize class or array type variables.

Next, the code to actually handle the appropriate work-sharing directive is inserted. At the end of the inner block, appropriate local variables associated to `lastprivate` and `reduction` variables are updated.

After the end of the inner block, a code to update the global copies of `lastprivate` and `reduction` variables is inserted. Only the thread performing the sequentially last iteration of the loop or the sequentially last section updates `lastprivate` variables. The master thread of the team updates `reduction` variables. Finally, the outer block is closed.

2.2.3.1.3.1 The `for` directive

Upon encountering a `for` directive, the compiler inserts code to create two `LoopData` structures. One of these is initialized to contain the details of the whole loop, while the other is used to hold details of particular chunks. The generated code then repeatedly calls the appropriate `getLoop()` function for the selected schedule, executing the blocks it is given, until there are no more blocks. If a dynamic scheduling strategy was used, the `ticketer` is then reset. Any reductions are carried out, and if the `nowait` clause is not specified, the `doBarrier()` method is called.

If the `ordered` clause is specified on a `for` directive, then a call to `resetOrderer()` method is inserted immediately prior to the loop, at which point the value of the first iteration number is definitely known. Upon encountering an `ordered` directive, the compiler inserts a call to `startOrdered()` before the relevant block with the parameter being the current value of the loop counter. After the block is inserted a call to `stopOrdered()`, with the parameter being the next value the loop counter would take after its current value, during sequential execution.

2.2.3.1.3.2 The `sections` directive

Upon encountering a `sections` directive, the compiler inserts code that repeatedly requests a ticket from the `ticketer`, and executes a different section depending on the ticket number. When there are no sections left, the `ticketer` is reset. If the `nowait` clause is not specified, the `doBarrier()` method is called.

2.2.3.1.3.3 The `master` directive

Upon encountering a `master` directive, the compiler inserts code to execute the relevant block if and only if the `OMP.getThreadNum()`method returns 0.

2.2.3.1.3.4 The `single` directive

Upon encountering a `single` directive, the compiler inserts code to get a ticket from the `ticketer`, execute the relevant block if and only if the ticket is zero, and then reset the `ticketer`. If the `nowait` clause is not specified, the `doBarrier()` method is called.

2.2.3.1.4 Synchronization directives

Upon encountering a `critical` directive, the compiler creates a synchronized block, with a call to `getLockByName()`. Upon encountering a `barrier` directive, the compiler inserts a call to the `doBarrier()` method.

2.2.4 HPC Experimental Environment

This section describes the experimental environment used in this thesis to evaluate the proposed mechanisms when using multithreaded Java applications in HPC environments.

2.2.4.1 Java Grande Benchmarks

The Java Grande Benchmarks [85] is suite of benchmark tests that provides ways of measuring and comparing alternative Java execution environments in ways that are important to *Grande* applications. A *Grande* application is one which uses large amounts of processing, I/O, network bandwidth, or memory. They include not only applications in science and engineering but also, for example, corporate databases and financial simulations. These benchmarks can be found in three different versions (sequential, multithreaded and JOMP), with three different sizes (A, B and C). The experiments performed in this thesis use the JOMP version – size B.

The multithreaded version of the Java Grande benchmark suite is designed for parallel execution on shared memory multiprocessors. It is composed by the following applications:

- ✓ Section 1: Low level operations – Barrier, ForkJoin, Sync
- ✓ Section 2: Kernels – Crypt, LUFact, SOR, Series, Sparse
- ✓ Section 3: Large scale applications – MolDyn, MonteCarlo, RayTracer

The JOMP version of the Java Grande benchmark suite is an implementation of the multithreaded version using JOMP directives. The following applications compose this version:

- ✓ Section 2: Kernels – Crypt, LUFact, SOR, Series, Sparse
- ✓ Section 3: Large scale applications – Euler, MonteCarlo, RayTracer

A detailed description of each one of these benchmarks can be found in Appendix A.

2.2.4.2 Hardware & software platform

The experimental platform used to conduct the evaluation of the proposed mechanisms in HPC environments is based on the SGI Origin 2000 architecture [129] with 64 MIPS R10000 processors at 250 MHz running the Irix 6.5.8 operating system and the SGI Irix JVM version Sun Java Classic 1.2.2.

All the experiments in this HPC environment have been performed in the so-called *cpusets* in Irix. A *cpuset* consists of a set of dedicated processors in a multiprogrammed machine. However, although a number of processors are reserved for the applications running inside the *cpuset*, other resources (like the interconnection network or the memory) are shared with the rest of applications running in the system. This sharing can interfere the behavior of the applications running inside the *cpuset* and produce noticeable performance degradation, which is difficult to quantify (and predict), because it depends on the system load and the application characteristics (a memory intensive application will be more interfered than an application with low memory use). The experiments reveal that this degradation can reach 10% for individual executions. In this case, this effect can be attenuated incrementing the number of measurements and discarding anomalous values. But when executing the applications as a part of a workload, observed degradation is around 20%, due to the interferences with the other applications in the workload plus the interferences with the rest of applications running in the system.

2.3 Multithreaded Java Application Servers in e-Business Environments

2.3.1 Introduction

In the latter days, e-business applications are becoming commonplace in current web sites. Some Java programming language characteristics, such as its portability or its support for parallel and distributed computing, have encouraged Internet applications programmers to use Java when developing these applications. Therefore, it is common to find Internet servers written in Java hosting current web sites. Typically, these servers are multithreaded Java applications in charge of serving clients requesting for web content, where each client connection is assigned to a thread that is the responsible of attending the received requests in this connection.

The logic of e-business applications is typically implemented using dynamic web content (i.e. following J2EE specification [132]). A request asking for dynamic web content requires some processing in the server (e.g. computation, access to a database…) before sending the response to the client, while the server can directly respond a request asking for static web content (i.e. HTML pages and images) with the requested file. Applications containing dynamic web content can be referred as dynamic web applications. Next section presents an overview on this kind of applications architecture and implementation.

2.3.2 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, which in its turn uses to consist of a front-end web server, an application server and a back-end database. Figure 2.2 shows a simplified version of dynamic web applications architecture. The client tier is responsible of interacting with application users and to generate requests to be attended by the server. The server tier implements the logic of the application and is responsible of serving user-generated requests.

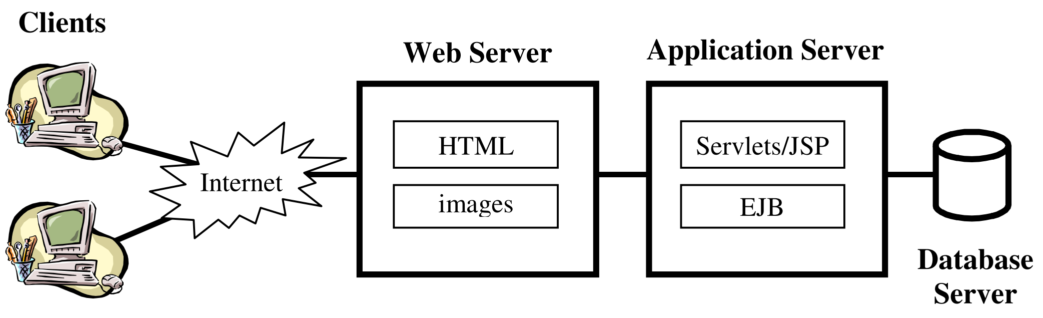


Figure 2.2. Dynamic web applications architecture

When the client sends to the web server an HTTP request for dynamic content, the web server forwards the request to the application server (as understood in this thesis, 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 by the web server as an HTTP response to the client. The implementation of the application logic in the application server may take various forms, including PHP [118], Microsoft Active Server pages [106], Java Servlets [136] and Enterprise Java Beans (EJB) [131].

This thesis focuses on Java Servlets, but the proposed mechanisms can be applied with the other mechanisms for generating dynamic web content, with the same effectiveness. A servlet is a Java class used to extend the capabilities of servers that 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.

2.3.3 Persistent HTTP Connections

As commented in the previous section, the Hypertext Transfer Protocol [55] (HTTP) allows servers and clients to send and receive data over the Internet. HTTP is a request and response protocol implemented over reliable TCP connections. In HTTP, it is always the client who initiates a transaction by establishing a connection and sending an HTTP request to the server, which processes this request and sends a response to the client. Either the client or the server can prematurely terminate a connection.

Prior to HTTP 1.1, whenever a client connected to a server, the connection was closed by the server right after the requested resource was sent. However, an Internet page can contain other resources, such as image files, applets, etc. Therefore, when a page is requested, the client also needs to download the resources referenced by the page. If the page and all resources it references are downloaded using different connections, the process will be very slow. That is why HTTP 1.1 introduced persistent connections. With a persistent connection, when a page is downloaded, the server does not close the connection straight away. Instead, it waits for the client to request all resources referenced by the page. This way, the page and referenced resources can be downloaded using the same connection. This saves a lot of work and time for the server, client and the network, considering that establishing and tearing down HTTP connections is an expensive operation.

2.3.4 Hosting Platforms

Typically, web applications run on hosting platforms that rent their resources to them. Applications owners pay for platform resources, and in return, the applications are provided with guarantees on resource availability and quality of service (which can be expressed in the form of a service level agreement [95][142] (SLA)). The hosting platform is responsible of providing sufficient resources to each application to meet its workload, or at least to satisfy the agreed QoS. Therefore, it is desirable that resources not used by some application may be distributed among other applications in the hosting platform.

Resource provisioning in a hosting platform can be based on either a dedicated or a shared model [7]. In the dedicated model, some cluster nodes are dedicated to each application and the provisioning technique must determine how many nodes to allocate to the application. In the shared model, which is the model considered in this thesis, node resources can be shared among multiple applications and the provisioning technique needs to determine how to partition resources on each node among competing applications.

Dedicated model used to be implemented as a cluster of servers where whole servers are distributed among web applications. Shared model can be implemented also as a cluster of servers where several applications can run in the same server, or using a multiprocessor machine for hosting all the applications. Clusters of servers are widely extended and are easily scalable but resource provisioning in these systems can be complex and inefficient. For example, traditional methods to switch a server from an underloaded to an overloaded application have entailed latencies of several minutes or more, due to software installation and configuration overheads [10]. In the same way, in session-based environments, transferring session state between servers is an inefficient task. As this thesis focus on e-commerce applications, which are typically session-based, and a dynamic provisioning mechanism able to react to unexpected workload changes in very short time is desired, the hosting platform is implemented using a multiprocessor machine.

Resource provisioning based on a shared model must consider an important issue. Since platform resources are shared by all the applications, when applications overload they can affect the performance of other applications. Consequently, a hosting platform should provide performance isolation, that is ensure that a minimal fraction of resources is available to serve requests from a certain application, and given a resource distribution between applications, an application should obtain the same performance independent of load generated by other applications.

2.3.5 Security in e-Business Applications

In e-business applications, all information that is confidential or has market value must be carefully protected when transmitted over the open Internet. These security capabilities between network nodes over the Internet are traditionally provided using HTTPS [125]. With HTTPS, which is based on using HTTP over SSL (Secure Socket Layer [56]), mutual authentication of both the sender and receiver of messages is performed to ensure message confidentiality.

2.3.5.1 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).

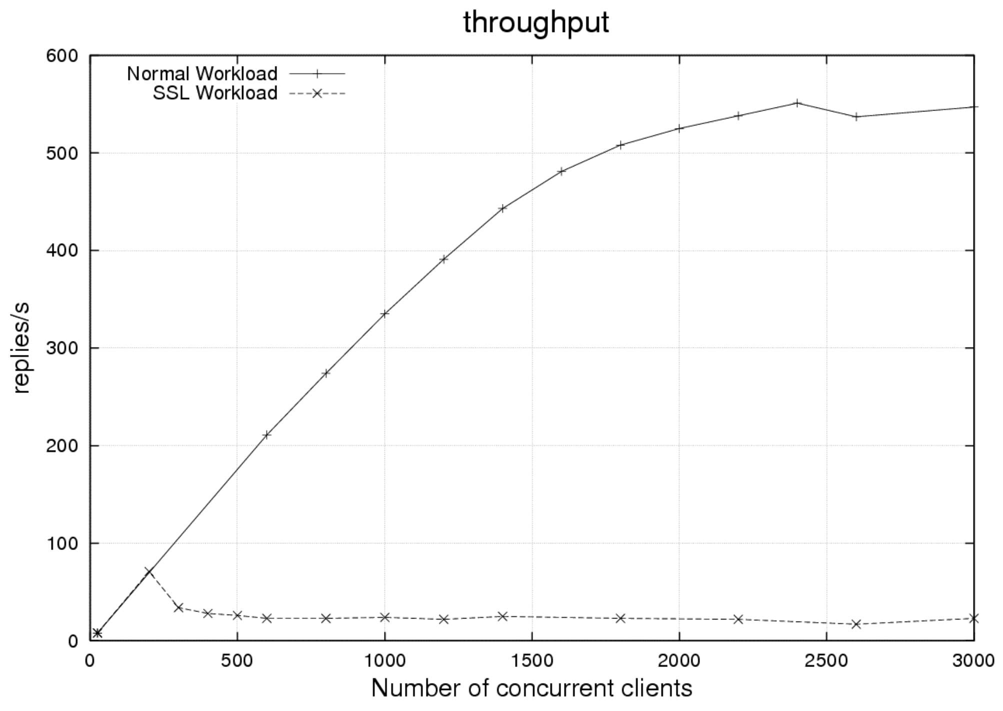


Figure 2.3. Tomcat scalability when serving secure vs. non-secure connections

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 increases the computation time necessary to serve a connection remarkably, due to the use of cryptography to achieve their objectives, becoming a CPU-intensive workload. This increment has a noticeable impact on server performance, which can be appreciated on Figure 2.3. This figure compares the throughput as a function of the number of clients obtained by a given application server when handling the same workload 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. On the other side, when using normal connections the maximum throughput is considerably higher (550 replies/s) and the server can scale until 1700 clients. Finally, notice also 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 impact of using SSL on server performance will be deeply discussed in Section 4.3.1 of this thesis.

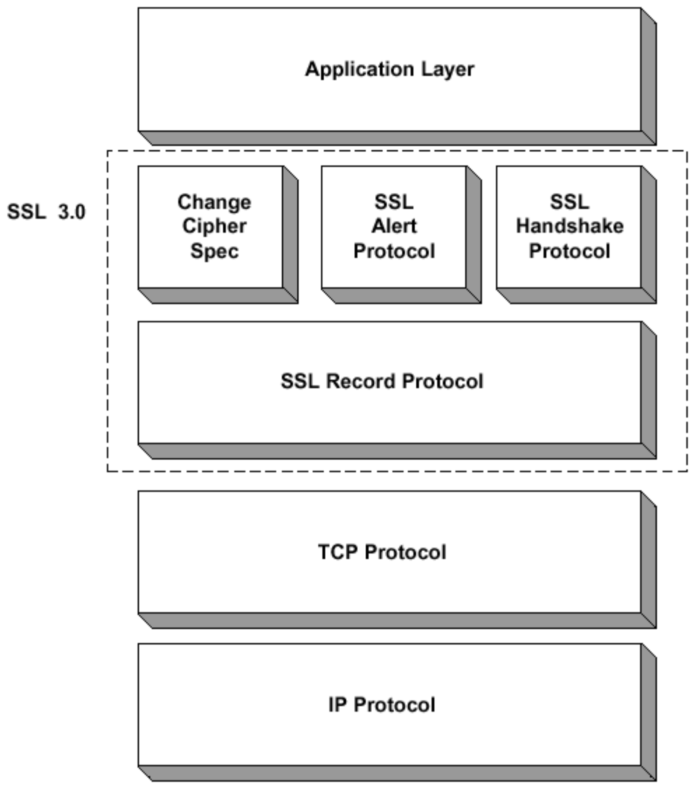


Figure 2.4. SSL protocol

The SSL protocol fundamentally has two layers of operation: the SSL handshake protocol and the SSL record protocol, as shown in Figure 2.4. Next subsection does an overview of these layers. The detailed description of the protocol can be found in RFC 2246 [47].

2.3.5.1.1 SSL Handshake protocol

The SSL Handshake protocol facilitates authentication of servers and clients, negotiation of the SSL session characteristics and data transfer. The server authenticates itself to the client using public-key techniques like RSA, and then the client and the server 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 2.5.

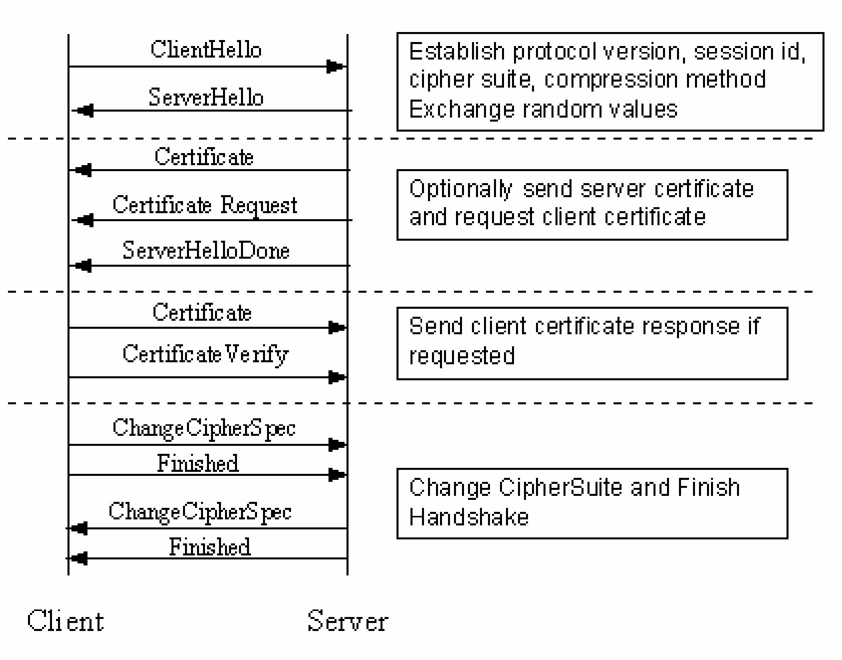

ClientHello
ServerHello
Establish protocol version, session id, cipher suite, compression method Exchange random values
Certificate
Certificate Request
ServerHelloDone
Optionally send server certificate and request client certificate
Certificate
CertificateVerify
Send client certificate response if requested
ChangeCipherSpec
Finished
ChangeCipherSpec
Finished
Change CipherSuite and Finish Handshake
Client
Server

Figure 2.5. SSL Handshake protocol negotiation

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. For example, the computational demand of a full SSL handshake in a 1.4 GHz Xeon machine is 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, reducing considerably the computation time for performing a resumed SSL handshake. For example, the computational demand of a resumed SSL handshake in a 1.4 GHz Xeon machine is around 2 ms. Notice the big difference between negotiate a full SSL handshake respect to negotiate 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).

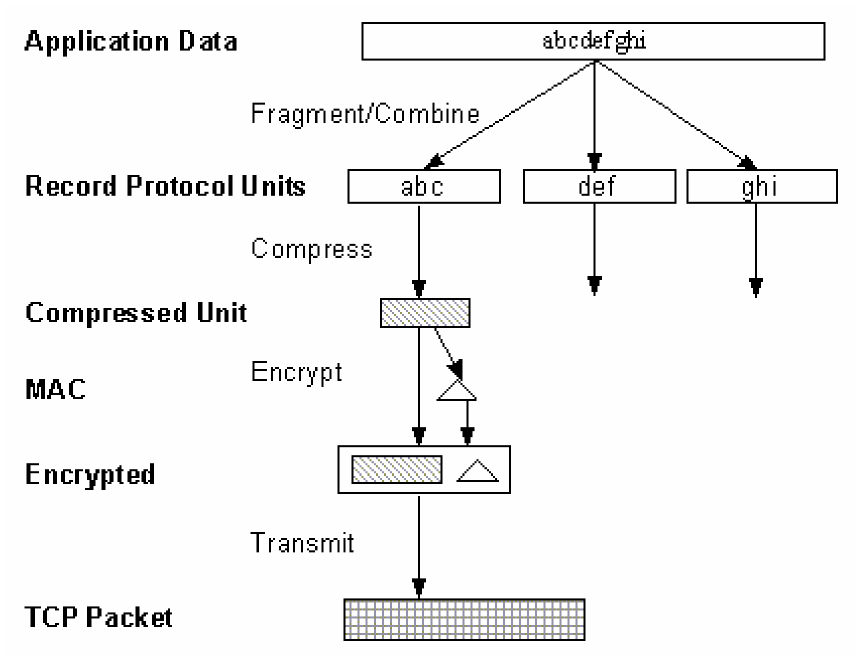


Figure 2.6. SSL Record protocol

2.3.5.1.2 SSL Record protocol

The SSL Record protocol permits the encapsulation of higher-level protocols, such as the SSL Handshake 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 sent into TCP packet. This process is detailed in Figure 2.6.

2.3.5.2 Java Secure Socket Extension (JSSE)

The Java Secure Socket Extension [135] (JSSE) is a set of classes that enable secure Internet communications. It implements a Java technology version of Secure Sockets Layer [56] (SSL) and Transport Layer Security [47] (TLS) protocols.

The JSSE package provides the `SSLSocket` and `SSLServerSocket` classes, which can be instantiated to create secure channels. The JSSE package supports the initiation of a handshake on a SSL connection in one of three ways. Calling `startHandshake` that explicitly begins handshakes, or any attempt to read or write application data through the connection causes an implicit handshake, or a call to `getSession` tries to set up a session if there is no currently valid session, and an implicit handshake is done. After handshaking has completed, session attributes can be accessed using the `getSession` method. If handshaking fails for any reason, the `SSLSocket` is closed, and no further communications can be done.

Notice that the JSSE package does not support any way to consult if an incoming SSL connection provides a reusable SSL session ID until the handshake is fully completed. Having this information prior to handshake negotiation could be very useful for example for servers in order to do overload control based on SSL connections differentiation, given the big difference existing on the computational demand of new SSL connections versus resumed SSL connections. It is important to notice that the verification about an incoming SSL connection provides a valid SSL session ID is already performed by the JSSE package prior handshaking in order to negotiate a full SSL handshake or a resumed SSL handshake. Therefore, the addition of a new interface to access this information would not involve additional cost.

2.3.6 e-Business Experimental Environment

This section describes the experimental environment used in this thesis to evaluate the proposed mechanisms when using multithreaded Java applications in e-business environments. The architecture of this experimental environment is shown in Figure 2.7.

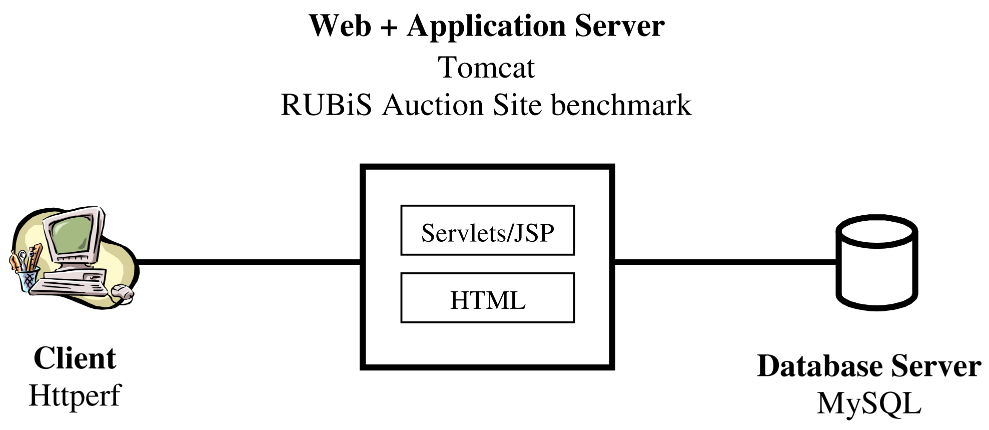


Figure 2.7. e-Business experimental environment

2.3.6.1 Tomcat servlet container

The experimental environment includes Tomcat [84] as the web and application 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 standalone server (serving both static and dynamic web content) or as a helper for a web server (serving only dynamic web content). This thesis uses Tomcat as a standalone server.

Tomcat follows a connection service schema where, at a given time, one thread (an HttpProcessor) is responsible of accepting a new incoming connection on the server listening port and assigning to it a socket structure. From this point, this HttpProcessor will be responsible of 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 previously created threads in order to avoid thread creation overheads. Persistent connections are a feature of HTTP 1.1 that allows serving different requests using the same connection, as commented in Section 2.3.3.

The pattern of a persistent connection in Tomcat is shown in Figure 2.8. On each *connection,* there is a distinction between the execution of several *requests* and the time devoted to maintain the connection persistence (*connection (no request)*), where server is maintaining opened the connection waiting for another client request. A connection timeout is programmed to close the connection if no more requests are received. For example, in this figure three different requests are served through the same connection. Notice that within every *request* is distinguished the *service* (execution of the servlet implementing the demanded request) from the *request (no service)*. This is the pre and post process that Tomcat requires to invoke the servlet that implements the demanded request.

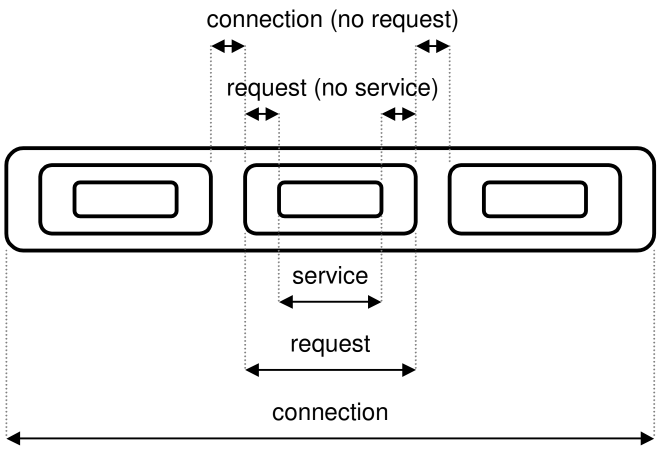


Figure 2.8. Tomcat persistent connection pattern

Figure 2.9 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 if a SSL Session ID is reused) as shown in Figure 2.9. For instance, if a client must establish a new HTTP connection because the server has closed its current HTTP connection due to connection persistence timeout expiration, as it reuses the underlying SSL connection, it negotiates a resumed SSL handshake.

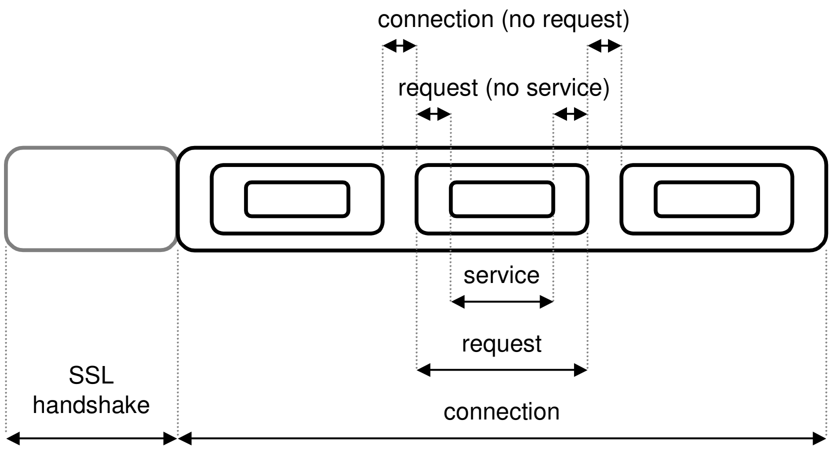


Figure 2.9. Tomcat secure persistent connection pattern

For the experiments in this thesis, Tomcat has been configured setting the maximum number of HttpProcessors to 100 and the connection persistence timeout to 10 seconds.

2.3.6.2 Auction site benchmark (RUBiS)

The experimental environment also includes a deployment of the RUBiS (Rice University Bidding System) [4] benchmark servlets version on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. RUBiS supplies implementations using some mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJB.

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. Table 2.1 shows the CPU demand distinguishing the time spent on each phase of the connection (measured in a 1.4 GHz Xeon machine) and the database demand (measured in a 2.4 GHz Xeon machine) for the RUBiS interactions used in this thesis (the read-only interactions). Notice that interactions requesting static web content do not spend any time in the database and CPU demand of interactions requesting dynamic web content is considerably larger than CPU demand of interactions requesting static web content.

Table 2.1. CPU and database demands of RUBiS interactions

	Service CPU Demand (us)	**Service Database Demand (us)**	**Request (no Service) CPU Demand (us)**	**Connection (no Request) CPU Demand (us)**
index.html	200	0	3,030	2,980
browse.html	188	0		
bid_now.jpg	167	0		
buy_it_now.jpg	180	0		
RUBiS_logo.jpg	3,650	0		
BrowseCategories	3,775	346		
BrowseRegions	5,655	396		
SearchItemsByCategory	2,810	18,235		
SearchItemsByRegion	2,280	6,234		
ViewItem	2,120	675		
ViewBidHistory	5,080	5,343		
ViewUserInfo	11,700	5,845		

2.3.6.3 Httperf

The client workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf [107]. 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, whose behavior is characterized with a complete set of performance measurements returned by Httperf. The configuration parameters of the tool used for the experiments presented in this thesis were set to create a realistic workload, with non-uniform reply sizes, sustaining 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. 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. RUBiS client emulator defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions. 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 allows also configuring a client timeout. If this timeout is elapsed 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. For the experiments in this thesis, Httperf has been configured setting the client timeout value to 10 seconds.

2.3.6.4 Hardware & software platform

The experimental platform used to conduct the evaluation of the mechanisms and policies proposed in e-business environments has been summarized in Table 2.2, specifying the sections of this thesis in which each configuration is used.

Table 2.2. Experimental platform used to evaluate the mechanisms proposed in e-business environments

		Section 3.4.2	**Section 4.3.1**	**Section 4.3.2**	**Section 5.3**
Client		RUBiS 1.4 client emulator 850 clients Experiment time = 150 sec Browsing mix 2 x 2-way Intel XEON 2.4 GHz, 2 GB RAM, 2.4 Linux kernel	Httperf 0.8 Client timeout = 10 s Experiment time = 10 min Browsing mix 2-way Intel XEON 2.4 GHz, 2 GB RAM, 2.6 Linux kernel		Httperf 0.8.5 Client timeout = 10 s Experiment time = 10 min Browsing mix 2-way Intel XEON 2.4 GHz, 2 GB RAM, 2.6 Linux kernel
Web + Application Server		Tomcat 4.0.6 RUBiS 1.4 Auction Site benchmark maxProcessors = 25, connectionTimeout = 10 s Sun JVM 1.4.2 Server JVM, -Xms, -Xmx = 512 MB 4-way Intel XEON 1.4 GHz, 2 GB RAM, 2.5 Linux kernel	Tomcat 5.0.19 RUBiS 1.4.2 Auction Site benchmark maxProcessors = 100, connectionTimeout = 10 s Sun JVM 1.4.2 Server JVM, -Xms, -Xmx = 1024 MB Common RSA-3DES-SHA cipher suit (RSA key 1024 bits) 4-way Intel XEON 1.4 GHz, 2 GB RAM, 2.6 Linux kernel		
Database server		MySQL v3.23.43 MM.MySQL v3.0.8 JDBC driver 2-way Intel XEON 2.4 GHz, 2 GB RAM, 2.4 Linux kernel	MySQL v4.0.18 MM.MySQL v3.0.8 JDBC driver 2-way Intel XEON 2.4 GHz, 2 GB RAM, 2.6 Linux kernel		
Network	**Client - Server**	Ethernet 100 Mbps	Ethernet 1 Gbps		
	Server - Database	Ethernet 100 Mbps		Ethernet 1 Gbps	

CHAPTER 3
ANALYSIS AND VISUALIZATION OF MULTITHREADED JAVA APPLICATIONS

3.1 Introduction

Previous experience on parallel applications has demonstrated that tuning this kind of applications for performance is mostly responsibility of (experienced) programmers [93]. Therefore, the performance analysis of multithreaded Java applications can be a complex work due to this inherent difficulty of analyzing parallel applications as well as the extra complexity added by the presence of the JVM. In this scenario, performance analysis and visualization tools that provide detailed information of multithreaded Java applications behavior are necessary in order to help users in the process of tuning the applications on the target parallel systems and JVM.

In the same way, the increasing load that the applications currently developed for Internet must support, demands new performance requirements to the Java application servers that host them. To achieve these performance requirements, fine-grain tuning of these servers is needed, but this tuning can be a hard work due to the large complexity of these environments (including the application server, distributed clients, a database server, etc.). Tuning Java application servers for performance requires also of tools that allow an in-depth analysis of application server behavior and its interaction with the other system elements. These tools must consider all levels involved in the execution of web applications (operating system, JVM, application server and application) if they want to provide significant performance information to the administrators (the origin of performance problems can reside in any of these levels or in their interaction).

Although a number of tools have been developed to monitor and analyze the performance of multithreaded Java applications (see Section 6.1), none of them allow a fine-grain analysis of the applications behavior considering all levels involved in the application execution. The main contribution in the “Analysis and Visualization of Multithreaded Java Applications” work area of this thesis is the proposal of a performance analysis framework to perform a complete analysis of the Java applications behavior based on providing to the user detailed and correlated information about all levels involved in the application execution, giving him the chance to construct his own metrics, oriented to the kind of analysis he wants to perform. The different levels considered by this performance analysis framework are shown in Figure 3.1.

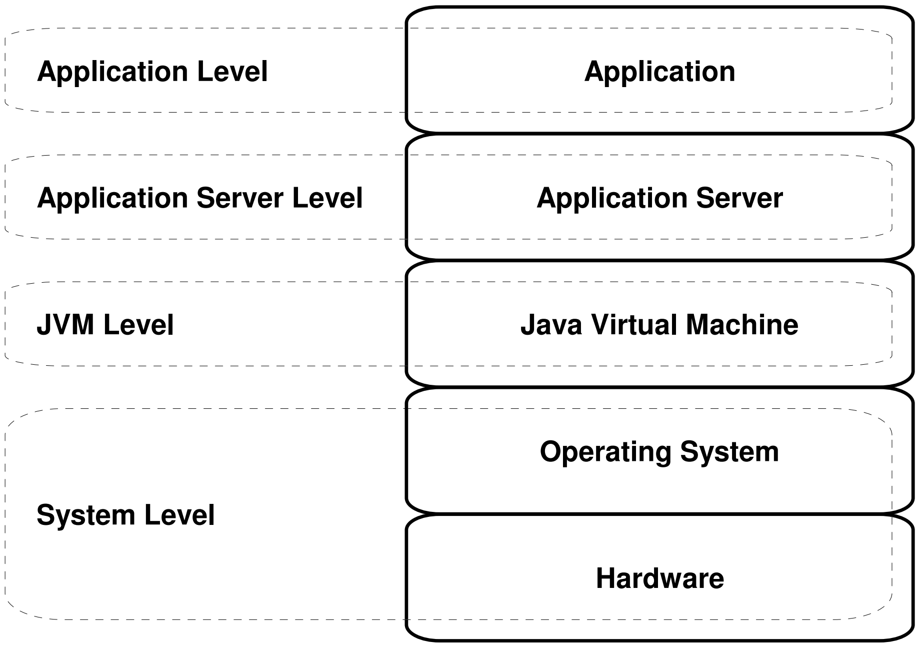


Figure 3.1. Instrumentation levels considered by the performance analysis framework

The performance analysis framework consists of two tools: an instrumentation tool, called JIS (Java Instrumentation Suite), and an analysis and visualization tool, called Paraver [116]. When instrumenting a given application, JIS generates a trace in which the information collected from all levels has been correlated and merged. The trace reflects the activity of each thread in the application recorded in the form of a set of predefined state transitions (that are representative of the parallel execution) and the occurrence of some predefined events. Later, the trace can be visualized and analyzed with Paraver (qualitatively and quantitatively) to identify the performance bottlenecks of the application.

3.2 Instrumentation Tool: JIS

The instrumentation tool (JIS) is responsible of collecting detailed information from all levels involved in the execution of Java applications. JIS correlates and merges this information in a final trace using the services provided by an instrumentation library. The next sections describe this library and the implementation of the different instrumentation levels considered by JIS.

3.2.1 Instrumentation Library

The proposed performance analysis framework use traces from real executions in the parallel target architecture in order to analyze multithreaded Java applications behavior. These traces reflect the activity of each thread in the application. This activity is recorded in the form of a set of predefined state transitions (that are representative of the parallel execution) and the occurrence of some predefined events.

The generation of these traces is supported by an instrumentation library that provides all the services required to generate traces. The library is implemented in C and, if necessary, could be invoked from Java through the Java Native Interface (JNI) [134]. JNI is a Java standard interface for invoking native code inside the Java code. The instrumentation library offers the following services:

- `ChangeState` - Change the state of a thread.
- `PushState` - Store the current state of a thread in a private stack and change to a new one.
- `PopState` - Change the state of a thread to the one obtained from the private stack.
- `UserEvent` - Emit an event (type and associated value) for a thread.

The library also offers combined services to change the state and emit an event: `ChangeandEvent`, `PushandEvent` and `PopandEvent`. Two additional services are offered to initialize and finish the instrumentation process:

- `InitLib` - Initialize the library internal data structures to start a parallel trace receiving as parameters: 1) the maximum number of threads participating in the execution, 2) the maximum amount of memory that the library has to reserve for each thread buffer, and 3) the mechanism used to obtain timestamps.

- `CloseLib` - Stop the tracing; this call makes the library dump to disk all buffered data not yet dumped and write resulting sorted trace to a file.

For each action being traced, the instrumentation library internally finds the time at which it was done. Timestamps associated to transitions and events can be obtained using generic timing mechanisms (such as the `gettimeofday` system call) or platform-specific mechanisms (for instance the high-resolution memory-mapped clock). All this data is written to an internal buffer for each thread (i.e. there is no need for synchronization locks or mutual exclusion inside the parallel tracing library). The data structures used by the tracing environment are also arranged at initialization time in order to prevent interference among threads (basically, to prevent false sharing). The user can specify the amount of memory used for each thread buffer. When the buffer is full, the instrumentation library automatically dumps it to disk.

When the application exits, the instrumentation library generates a trace file joining the per-thread buffers containing the information that has been collected from all levels. This information is then correlated and merged. This adds an extra overhead to the whole execution time of the application that does not have any impact in the trace.

3.2.2 System Level

The JIS instrumentation at the system level can obtain information of the threaded execution of the application inside the operating system by providing the threads state along time and the system calls issued (I/O, sockets, memory management and thread management). This is the only level where the instrumentation depends on the underlying platform. In this thesis, two implementations of the instrumentation at the system level have been performed:

- A dynamic interposition mechanism that obtains information about the supporting threads layer (i.e. Pthreads library [121]) without recompilation has been implemented for the SGI Irix platform.
- A device driver that gets information from a patched Linux kernel has been developed for the Linux platform.

3.2.2.1 SGI Irix platform

The JIS instrumentation at the system level in the SGI Irix platform can provide information about the supporting threads layer (i.e. Pthreads library), mutual exclusion and synchronization primitives (mutexes and conditional variables) and system calls issued (I/O, sockets and thread management).

The information acquisition at this level is accomplished by dynamically interposing the instrumentation code at run time using DITools [126]. This dynamic code interposition mechanism allows JIS not to require any special compiler support and makes unnecessary to rebuild neither the bytecode of the application nor the executable of the JVM.

3.2.2.1.1 System level information

As commented before, JIS instrumentation at the system level provides information about threads state. Table 3.1 summarizes the different states that JIS instrumentation at the system level in the SGI Irix platform considers for a thread.

Table 3.1. Thread states considered by the JIS instrumentation at the system level in the SGI Irix platform

STATE	**DESCRIPTION**
INIT	Thread is being created and initialized
READY	Thread is ready for running, but there is no CPU available
RUN	Thread is running
BLOCKED IN CONDVAR	Thread is blocked waiting on a monitor
BLOCKED IN MUTEX	Thread is blocked waiting to enter in a monitor
BLOCKED IN I/O	Thread is blocked waiting for an I/O operation
STOPPED	Thread has finalized

The required knowledge about the execution environment can be expressed using a state transition graph, in which each transition is triggered by a procedure call and/or a procedure return. Figure 3.2 and Figure 3.3 present the state transition graphs for both execution models[1] (green and native threads, respectively) supported by JIS instrumentation at the system level in the SGI Irix platform, in which nodes represent

[1] Some implementations of the JVM (e.g. SGI Irix JVM) allow Java threads to be scheduled by the JVM itself (the so-called green threads model) or by the operating system (the so-called native threads model). When using green threads, the operating system does not know anything about threads that are handled by the JVM (from its the point of view, there is a single process and a single thread). In the native threads model, threads are scheduled by the operating system that is hosting the JVM.

states, and edges correspond to procedure calls (indicated by a + sign) or procedure returns (indicated by a - sign) causing a state transition.

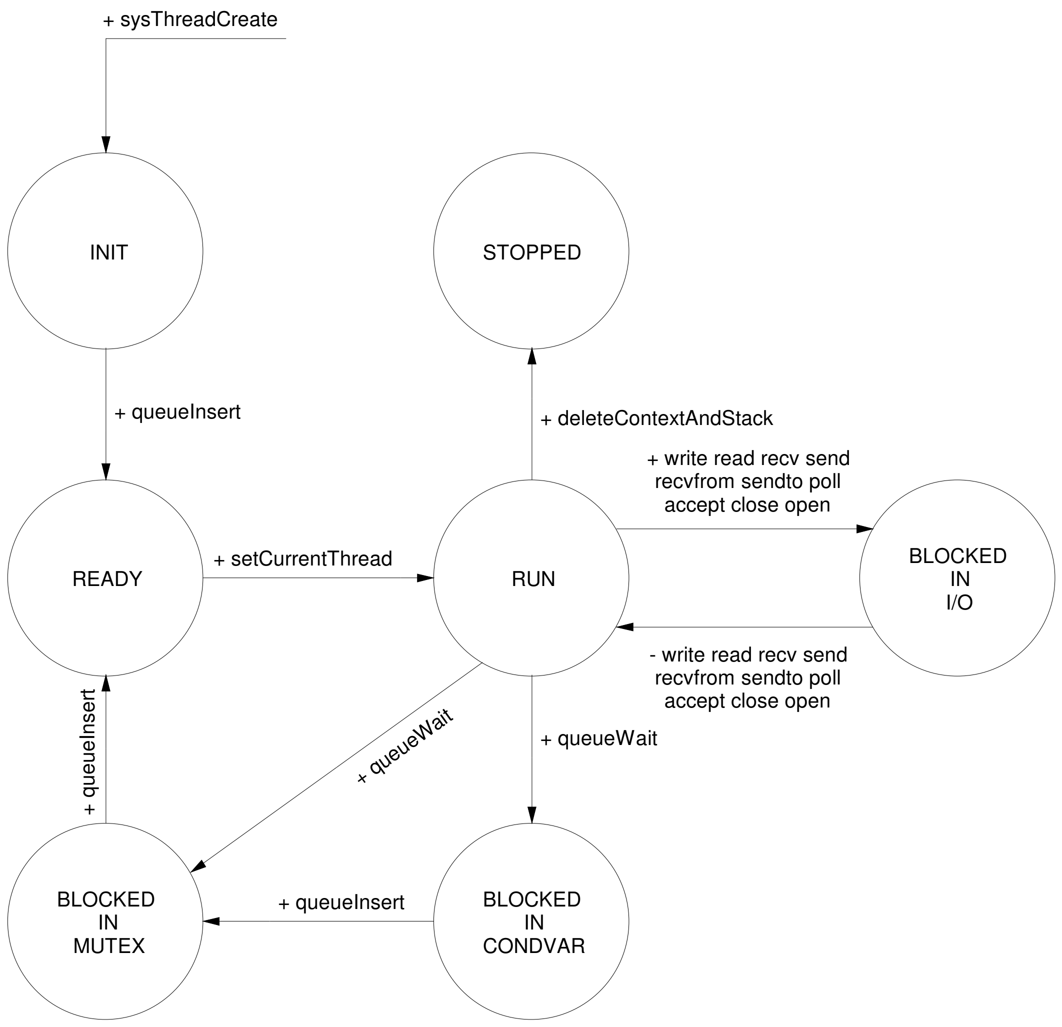


Figure 3.2. State transition graph for green threads considered by the JIS instrumentation at the system level in the SGI Irix platform

These transition graphs are then used to derive the interposition routines used to keep track of the state in the instrumentation backend. These routines are simple wrappers of functions that change the thread state, emit an event and/or save thread information in the internal structures of JIS using the services offered by the instrumentation library described in Section 3.2.1. These wrappers can perform instrumentation actions before (`_PRE`) and/or after (`_POST`) the call being interposed. Figure 3.4 shows a simple example of procedure wrapper and the skeleton of the function executed before the activation of function `pthread_cond_wait`.

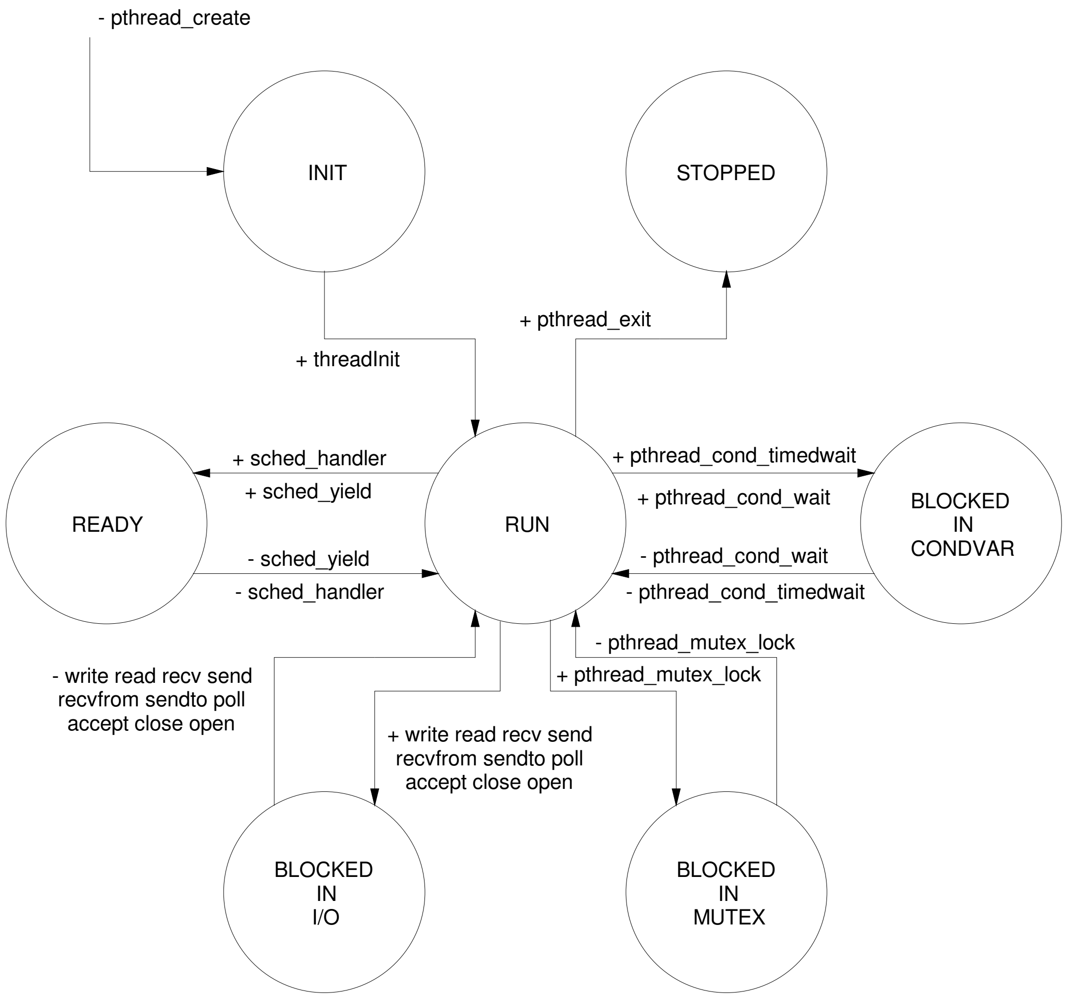


Figure 3.3. State transition graph for native threads considered by the JIS instrumentation at the system level in the SGI Irix platform

```
int pthread_cond_wait_wrapper (pthread_cond_t *p, pthread_mutex_t *m) {
    pthread_cond_wait_PRE ((long)p,(long)m);
    ret = pthread_cond_wait (p,m);
    pthread_cond_wait_POST ((long)p,(long)m);
    return ret;
}

void pthread_cond_wait_PRE (long condvar_id, long mutex_id) {
  pth_id = pthread_self();
  /* find Paraver thread identifier (jth_id = 1 .. n) of pth_id */
  PushandEvent(jth_id, BLOCKED_IN_CONDVAR, EVENT_BLOCKED_IN_CONDVAR,
             condvar_id);
  /* update internal structures */
}
```

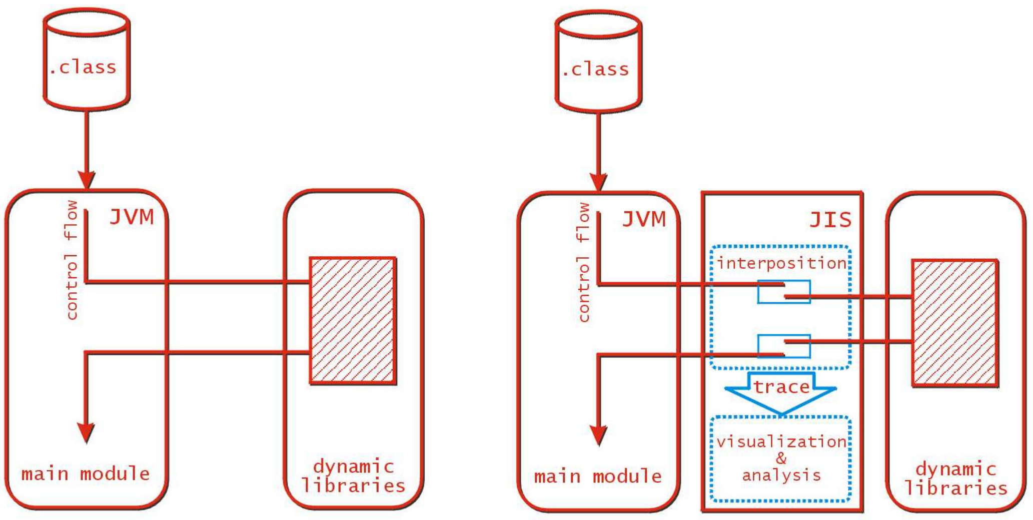
Figure 3.4. Example of procedure wrapper

JIS instrumentation at the system level in the SGI Irix platform complements the information of threads state by generating events that indicate:

- ✓ The operations related to mutual exclusion (`lock`, `trylock`, `locked`, `unlock`) or thread synchronization on conditional variables (`wait`, `waited`, `signal`, `broadcast`).
- ✓ The system calls performing socket operations (`accept`, `send`, `recv`, `sendto`, `recvfrom`, `close`).
- ✓ The system calls performing I/O operations (`open`, `read`, `write`, `close`, `poll`).
- ✓ The system calls performing thread operations (`sched_yield`, `sleep`).
- ✓ In which kernel threads are executing the Java threads.

3.2.2.1.2 Dynamic code interposition

Dynamic linking is a feature available in many modern operating systems. Program generation tools (compilers and linkers) support dynamic linking via the generation of linkage tables. Linkage tables are redirection tables that allow delaying symbol resolution to run time. At program loading time, a system component fixes each pointer to the right location using some predefined resolution policies. Usually, the format of the object file as well as these data structures are defined by the system Application Binary Interface (ABI). The standardization of the ABI makes possible to take generic approaches to dynamic interposition.



(a) *(b)*

Figure 3.5. Dynamic code interposition

The instrumentation methodology is based on the fact that the JVM invokes a set of run-time services at key places in order to use threads or to synchronize them.

These services are dynamic linked with the JVM via the use of linkage tables, like is shown in Figure 3.5.a. The interposition mechanism modifies the appropriate linkage table entries in order to redirect references to instrumentation wrappers, as shown in Figure 3.5.b. These wrappers track state changes and issue events by invoking services of the instrumentation library.

3.2.2.1.3 Instrumentation overhead

The overhead of the JIS instrumentation at the system level in the SGI Irix platform is determined using the LUAppl application, which is a LU reduction kernel over a two-dimensional matrix of double-precision elements taken from [111]. The results of the overhead measurement when instrumenting the LUAppl are shown in Table 3.2. The table reports the execution time in milliseconds of the original LUAppl with respect to the LUAppl when instrumenting its behavior, when running with 4 threads and different problem sizes. Notice that, the overhead is kept reasonably low (below 8%) and considered acceptable taking into account the level of detail provided by the process.

Table 3.2. Overhead of the JIS instrumentation at the system level in the SGI Irix platform for LUAppl

Matrix size	**Original**	**Instrumented**	**Overhead**
128x128	2795	2996	7.19 %
256x256	17542	17975	2.47 %
512x512	109976	110857	0.80 %

3.2.2.2 Linux platform

The JIS instrumentation at the system level in the Linux platform can provide information about the threads state and the system calls issued (I/O, sockets, memory management and thread management). This information is directly extracted from inside kernel using two different layers: one based in a kernel source code patch and the other in a system device and its corresponding driver (implemented in a Linux kernel module).

3.2.2.2.1 System level information

The different states that JIS instrumentation at the system level in the Linux platform considers for a thread are summarized in Table 3.3. Notice that, this is not the complete list of possible thread states on Linux. Other states are not considered relevant to study the behavior of multithreaded Java applications in parallel environments.

Table 3.3. Thread states considered by the JIS instrumentation at the system level in the Linux platform

STATE	**DESCRIPTION**
READY	Thread is ready for running, but there is no CPU available
RUN	Thread is running
BLOCKED	Thread is blocked

Figure 3.6 shows the state transition graph supported by JIS instrumentation at the system level in the Linux platform, in which nodes represent states, and edges correspond to procedure calls causing a state transition. This transition graph is used to derive the interposition routines used to keep track of the state in the instrumentation backend. These routines are simple wrappers of functions that change the thread state, emit an event and/or save thread information in the internal structures of JIS using the services offered by the instrumentation library.

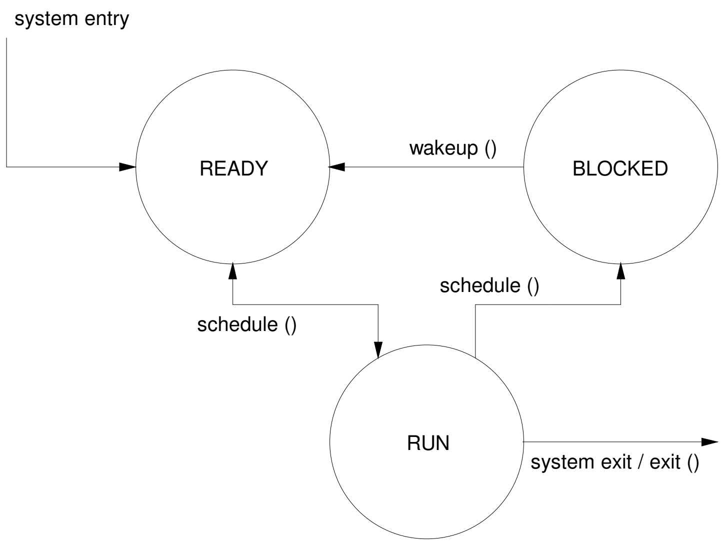


Figure 3.6. State transition graph considered by the JIS instrumentation at the system level in the Linux platform

JIS instrumentation at the system level in the Linux platform complements the information of threads state by generating events that indicate:

✓ The system calls performing I/O operations (`open`, `close`, `read`, `write`, `lseek`, `poll`, `select`) with their associated entry and exit timestamps and the size and result of the performed operations.

- ✓ The system calls performing socket operations (`socket`, `accept`, `bind`, `sendto`, `recvfrom`) with their associated entry and exit timestamps and the size and result of the performed operations.
- ✓ The system calls performing memory operations (`brk`, `mmap`, `munmap`, `mprotect`, `madvise`) with their associated entry and exit timestamps and the size and result of the performed operations.
- ✓ The system calls performing thread operations (`sched_yield`, `nanosleep`) with their associated entry and exit timestamps and the size and result of the performed operations.
- ✓ In which processors are executing the Java threads.

3.2.2.2.2 Kernel source code patch

Some system events cannot be extracted by any other way than inserting hooks inside the kernel source. These special events are related to kernel threads state and other ways of obtaining this information are not enough. For instance, Linux offers an interesting way to extract process status on system: the `proc` file system. The problem comes with the way this system interface divides the two main process states: `Runnable` and `Blocked`. `Runnable` implies that a process is ready to run on a processor, but does not give information about if it is really running or if it is waiting for a processor to start execution. This issue makes the `proc` file system insufficient to determine thread status at each moment in time. Thus, a kernel patch has to be used to obtain information about the state of the threads of the system at each moment in time. This information is obtained directly from the scheduler routine and notified to an instrumentation driver.

Information from system calls is obtained by intercepting some entries of the system call table. The global system call table is modified in order to generate notifications to the instrumentation driver and invoke the original system call function in order to preserve the original system behavior.

3.2.2.2.3 Instrumentation device driver

The instrumentation driver receives the notifications from the patched Linux kernel when a thread state change is produced or an intercepted system call is invoked. This driver requires a device that controls it. The device driver is implemented inside a Linux kernel module and is responsible of attending the notifications received from the patched kernel by tracking state changes and issuing events using the services of the instrumentation library. The device driver implements also basic functions operable over the device and to allocate the system events buffer. Basic implemented functions are: `open`, `close`, `ioctl` and `mmap`.

`Open` and `close` calls are used to be able to work with the device. `Ioctl` call is used to control the system space instrumentation from the user space code. When the instrumented application finishes its execution, the shared library controlling the instrumentation can use the `ioctl` call to indicate to the kernel module that the instrumentation process is concluded. Finally, the `mmap` call is implemented to allow the user space instrumentation code to work transparently with the system space buffer and be able to merge both event buffers, system and space one, into a unique final trace. A diagram summarizing the architecture of the JIS instrumentation at the system level in the Linux platform is shown in Figure 3.7.

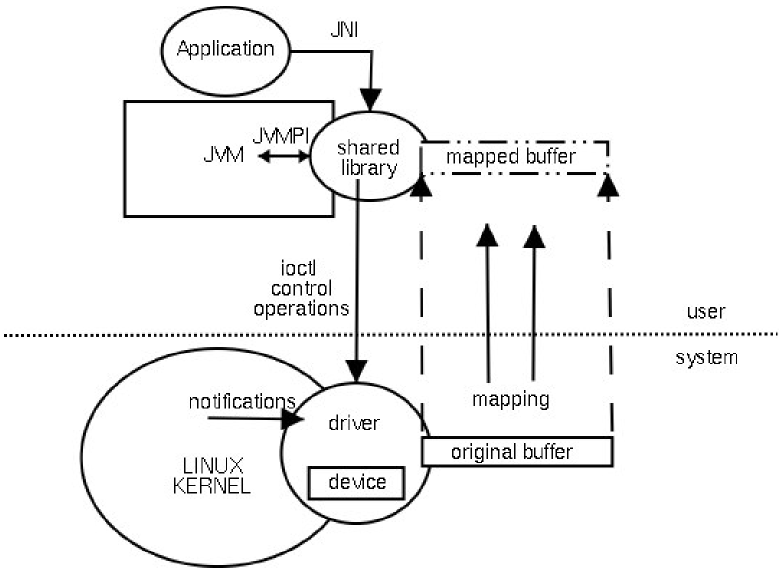


Figure 3.7. Architecture of the JIS instrumentation at the system level in the Linux platform

3.2.2.2.4 Instrumentation overhead

The overhead of the JIS instrumentation at the system level in the Linux platform is determined using the LUAppl application, which has been introduced in Section 3.2.2.1.3. The results of the overhead measurement when instrumenting the LUAppl are shown in Table 3.4. The table reports the execution time in milliseconds of the original LUAppl with respect to the LUAppl when instrumenting its behavior, when running with 4 threads and different problem sizes. Notice that, the overhead is kept very low (below 4%), considered acceptable in order to not to affect the conclusions extracted from applications analysis.

Table 3.4. Overhead of the JIS instrumentation at the system level in the Linux platform for LUAppl

Matrix size	Original	Instrumented	Overhead
250x250	699	722	3.29 %
500x500	3434	3450	0.47 %
750x750	9478	9492	0.15 %
1000x1000	20662	20710	0.23 %

3.2.3 JVM Level

The JIS instrumentation at the JVM level can obtain information about JVM internals, considering Java abstractions like classes, objects, methods, threads and monitors. The information acquisition at the JVM level is accomplished by using the Java Virtual Machine Profiler Interface [143] (JVMPI). JVMPI is a common interface that can be used to obtain profiling information from the running Java application by introducing hooks inside the JVM code in order to be notified about some predefined Java events. Using JVMPI, there is no need to change the source of the application or recompile it, only is necessary to include an option to the Java interpreter. However, the use of JVMPI can result in severe overheads, because of the high notification frequency of some JVMPI events (e.g. method entry and method exit events). For these reason, JIS only obtains at this level the name of the Java threads and information about the operations performed on the different Java Monitors (wait, notify, notifyAll, contended enter, contended exit).

3.2.3.1 JVMPI

The JVMPI is based on the idea of creating a shared library that is dynamically linked with the JVM if the user passes an instrumentation option to the Java interpreter. This library will be notified about selected internal JVM events. Choosing hooked events is done at JVM load time using a standard implemented method on the library that is invoked by the JVM. This method is called `JVM_OnLoad`. An example of selecting events that have to be notified in this method is shown in Figure 3.8. In this example, the notification of waits in a monitor is enabled

(`JVMPI_EVENT_MONITOR_WAIT` event). The `JVM_OnLoad` function also specifies the routine that has to be called each time that a requested event is produced, in this example the `notifyEvent` routine.

When a selected event is produced, this event is notified through a call to the `notifyEvent` function that can determine, by parsing received parameters, what event is taking place. Depending on this, the function will track the state changes or will issue the necessary events using the services provided by the instrumentation library.

```
#include <jvmpi.h>

// global jvmpi interface pointer
static JVMPI_Interface *jvmpi_interface;

// function for handling event notification
void notifyEvent(JVMPI_Event *event) {
  switch(event->event_type) {
    ...
    case JVMPI_EVENT_MONITOR_WAIT:
    ...
  }
}

// profiler agent entry point
JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *jvm, char *options, void
*reserved) {

    // get jvmpi interface pointer
    if ((jvm->GetEnv((void **)&jvmpi_interface, JVMPI_VERSION_1)) < 0)
    {
      fprintf(stderr, "Error in obtaining jvmpi interface pointer\n");
      return JNI_ERR;
    }

    // initialize jvmpi interface
    jvmpi_interface->NotifyEvent = notifyEvent;

    // enabling class load event notification
    jvmpi_inter->EnableEvent(JVMPI_EVENT_MONITOR_WAIT, NULL);

    return JNI_OK;
}
```

Figure 3.8. JVMPI initialization

The JVMPI comes with the standard Java SDK from version 1.2. However, new SDK release 1.5, has replaced JVMPI with a new profiling interface, the JVM Tool Interface [137] (JVMTI).

3.2.4 Application Server Level

Information about the internals of the application server can be obtained at the application server level. This information is reported using events that indicate the begin/end of services (i.e. servlets and EJB), requests, HTTP connections, SSL connections, database transactions, etc.

The information acquisition at this level is accomplished by injecting instrumentation probes at specific points in the application server where events are required to signal server relevant actions. These probes invoke the services (which generate the events and/or state transitions) of the instrumentation library through the Java Native Interface (JNI). The instrumentation library is dynamically linked with the JVM and offers a common JNI interface to the Java applications. This allows the use of the instrumentation at the application level in all platforms supporting Java.

The instrumentation probes can be directly injected in the application server source code, if this is available. Otherwise, other techniques can be used to extract information from the application server level when source code is not available. For example, the Java Automatic Code Interposition Tool [57] (JACIT) allows adding code to already compiled classes without recompilation. JACIT is based on using Aspect programming techniques [60] to enable the work with code although source code is not available or to extend features from a closed product.

For example, Figure 3.9 shows the code injected in the `HttpServlet` class in order to obtain instrumentation information about when the services begin and end.

```
package javax.servlet.http;

public abstract class HttpServlet extends GenericServlet
    implements java.io.Serializable
{
  ...
  protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
  {
    bjs.UserEvent(SERVICE,BEGIN);
    ...
    doGet(req, resp);
    ...
    bjs.UserEvent(SERVICE,END);
  }
  ...
}
```

Figure 3.9. Code injection mechanism in the HttpServlet class

3.2.5 Application Level

JIS can provide also the user with information about the Java application level. The information acquisition at this level is accomplished in the same way that in the application server level (i.e. by injecting instrumentation probes that invoke the services of the instrumentation library using the JNI at specific points in the Java application where events are required to signal application relevant actions). As in the application server level instrumentation, the instrumentation at the application level can be used in all platforms supporting Java.

3.2.5.1 Instrumentation of JOMP applications

As a special case of instrumentation at the application level, support for JOMP applications [23] has been added to JIS. JOMP includes OpenMP-like extensions to specify parallelism in Java applications using a shared-memory programming paradigm. The instrumentation provides a detailed analysis of the parallel behavior at the JOMP programming model level. At this level, the user is faced with parallel, work-sharing and synchronization constructs.

3.2.5.1.1 JOMP programming model level information

Table 3.5 summarizes the different states that the instrumentation of JOMP applications considers for a thread. The `RUN` state corresponds to the execution of useful work, i.e. execution of work in the original source code. The `IDLE` state reflects the fact that a thread is waiting (outside a parallel region) for work to be executed. The JOMP runtime library creates threads at the first parallel region and keeps them alive until the end of the application. In the meanwhile, they check for new work to be executed, and if found, execute it. The `OVERHEAD` state shows that the thread is executing code associated with definition and initialization of `private`, `lastprivate`, `firstprivate` and `reduction` variables, or the determination of the tasks to be done in a work-sharing construct. The `SYNCH` state refers to the situation in which a thread is waiting for another thread to reach a specific point in the program, or for access to a `ticketer` to guarantee specific ordered actions.

Table 3.5. Thread states considered by JOMP applications instrumentation

STATE	DESCRIPTION
IDLE	Thread is waiting for work to be executed
RUN	Thread is running
OVERHEAD	Thread is executing JOMP overhead
SYNCH	Thread is synchronizing with other threads in the team

The instrumentation of JOMP applications can also report events that provide additional information about the JOMP constructs being executed. Each event has two fields associated: type and value. The type is used to indicate the entry/exit to/from a parallel, work-sharing or synchronization construct. The value is used to relate the event type with the source code (for instance, line number in the source code and method name). The communication between the event types and values assigned by the compiler and Paraver is done through a configuration file generated by the compiler itself.

3.2.5.1.2 Code injection

The information acquisition is accomplished in the same way as explained for the generic application level instrumentation, that is, by injecting instrumentation probes that invoke the services (which generate the events and/or state transitions) of the instrumentation library using the JNI at specific points in the JOMP application. The JOMP compiler has been modified in order to inject these probes in the JOMP application source code (where the state transitions occur and where events are required to signal JOMP relevant actions) without user intervention during the code generation phase.

Figure 3.10.b shows the instrumented parallel code for the simple example shown in Figure 3.10.a. Notice that the compiler forces a state change to `OVERHEAD` as soon as the master thread starts the execution of the block of code that encapsulates the parallel construct in the main method. The previous state is stored in an internal stack so that the master thread can restore it as soon as it finishes the execution of this block of code. When changing to `OVERHEAD` state, the master thread also emits an event with type `EVENT_PARALLEL_BEGIN` that indicates the beginning of the parallel construct and with value 500 indicates that this parallel construct is found at a certain line and method in the original source code. In the same way, when the master thread restores its previous state, it also emits an event with type `EVENT_PARALLEL_END` that indicates the end of the parallel construct and with value the same 500.

Each thread in the team executing the `go()` method changes to the `RUN` state when it starts the execution of the user code. After executing the original user code, each thread changes to the `OVERHEAD` state for managing reduction variables. Then the thread changes to the BLOCKED state and gets into a barrier. When all threads have reached the barrier, they restore their previous state.

```
public class Hello {
  public static void main (String argv[]) {
    int myid;
    //omp parallel private (myid)
    {
      myid = OMP.getThreadNum();
      System.out.println("Hello from" + myid);
    }
  }
}
```

(a) original code

```
public class Hello {
  public static void main (String argv[]) {
    int myid;
    bjs.InitLib(jomp.runtime.OMP.getMaxThreads());
    // OMP PARALLEL BLOCK BEGINS
    {
      bjs.PushandEvent(jomp.runtime.OMP.getThreadNum(),OVERHEAD,
                       EVENT_PARALLEL_BEGIN,500);
      __omp_Class0 __omp_Object0 = new __omp_Class0();
      __omp_Object0.argv = argv;
      try {
        jomp.runtime.OMP.doParallel(__omp_Object0);
      } catch(Throwable __omp_exception) {
        jomp.runtime.OMP.errorMessage();
      }
      argv = __omp_Object0.argv;
      bjs.PopandEvent(jomp.runtime.OMP.getThreadNum(),
                      EVENT_PARALLEL_END,500);
    }
    // OMP PARALLEL BLOCK ENDS
    bjs.CloseLib();
  }
}

// OMP PARALLEL REGION INNER CLASS DEFINITION BEGINS
private static class __omp_Class0 extends jomp.runtime.BusyTask {
  String [] argv;
  public void go(int __omp_me) throws Throwable {
    int myid;
    // OMP USER CODE BEGINS
    {
      bjs.PushState(jomp.runtime.OMP.getThreadNum(),RUN);
      myid = OMP.getThreadNum();
      System.out.println("Hello from" + myid);
      bjs.ChangeState(jomp.runtime.OMP.getThreadNum(),OVERHEAD);
    }
    // OMP USER CODE ENDS
    bjs.ChangeState(jomp.runtime.OMP.getThreadNum(),BLOCKED);
    jomp.runtime.OMP.doBarrier(__omp_me);
    bjs.PopState(jomp.runtime.OMP.getThreadNum());
  }
}
// OMP PARALLEL REGION INNER CLASS DEFINITION ENDS
```

(b) instrumented transformed code

Figure 3.10. Example of code injection made by the JOMP compiler: `parallel` directive

3.2.5.1.3 Instrumentation overhead

The overhead of the instrumentation of JOMP applications is determined using the LUJOMP application, which is a JOMP version of the LUAppl presented in Section 3.2.2.1.3. The code of LUJOMP is shown in Figure 3.11.

```
for (k=0; k < SIZE; k++) {
    //omp parallel
    {
        //omp for schedule(static) nowait
        for (int i=k+1; i < SIZE; i++) {
            matrix[i][k] = matrix[i][k] / matrix[k][k];
        }
        //omp for schedule(static)
        for (int i=k+1; i <SIZE; i++) {
            for (int j=k+1; j < SIZE; j++) {
                matrix[i][j] = matrix[i][j] - matrix[i][k] * matrix[k][j];
            }
        }
    }
}
```

Figure 3.11. Source code of JOMP version of LUAppl application

The results of the overhead measurement when instrumenting the LUJOMP are shown in Table 3.6. The table reports the execution time in milliseconds of the original LUJOMP with respect to the LUJOMP when instrumenting its behavior, when running with 4 threads and different problem sizes. Notice that the overhead is very low (less than 3%).

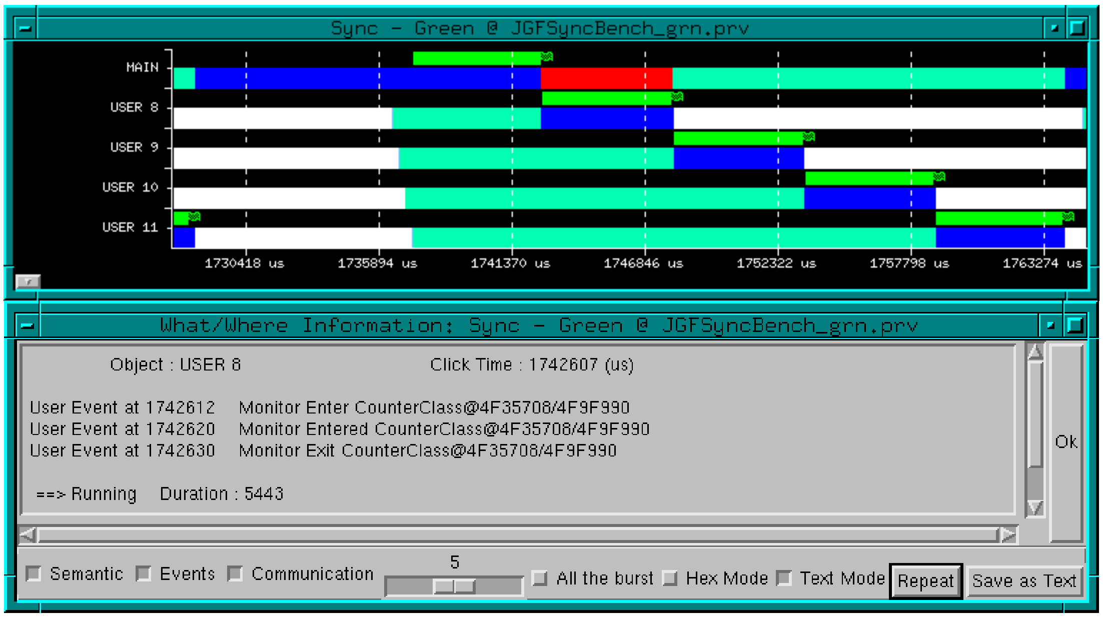
Table 3.6. Overhead of the JOMP applications instrumentation for LUAppl

Matrix size	Original	Instrumented	Overhead
128x128	1899	1949	2.63%
256x256	15842	16222	2.4%
512x512	105962	108092	2%

3.3 Visualization Tool: Paraver

Paraver [116] is a flexible trace visualization and analysis tool developed at CEPBA [33] based on an easy-to-use Motif GUI. Paraver was developed to respond to the need to have a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Large amount of information useful to improve the decisions on whether and where to invert the programming effort to optimize an application is accessible with Paraver. Trace analysis with Paraver ranges from the visualization of the graphical trace representation to the calculation of multiple statistics in order to detect performance problems. In any case, all the possible views and statistical calculations made on a trace file can be saved as Paraver configuration files for later reuse. It allows users to create a large amount of preset views of the trace file that can point out some performance indexes or conflictive situations in a direct way.

The graphical views of the trace files are based on the representation of threads, characterized by their state along time and by some punctual events. The combination of states and events makes possible to do a detailed and intuitive representation of an application behavior. A sample of the Paraver visualization module can be seen in the top part of Figure 3.12. On it, threads are represented on the vertical axis and the horizontal axis is used for the timeline. The color of each thread along time indicates its state. The state value of each thread can be obtained directly from the trace file or can be calculated as a function of the thread state and the event values. Textual information related to the states and the events can also be obtained with this view by clicking at any point in the trace, as shown in bottom part of Figure 3.12.



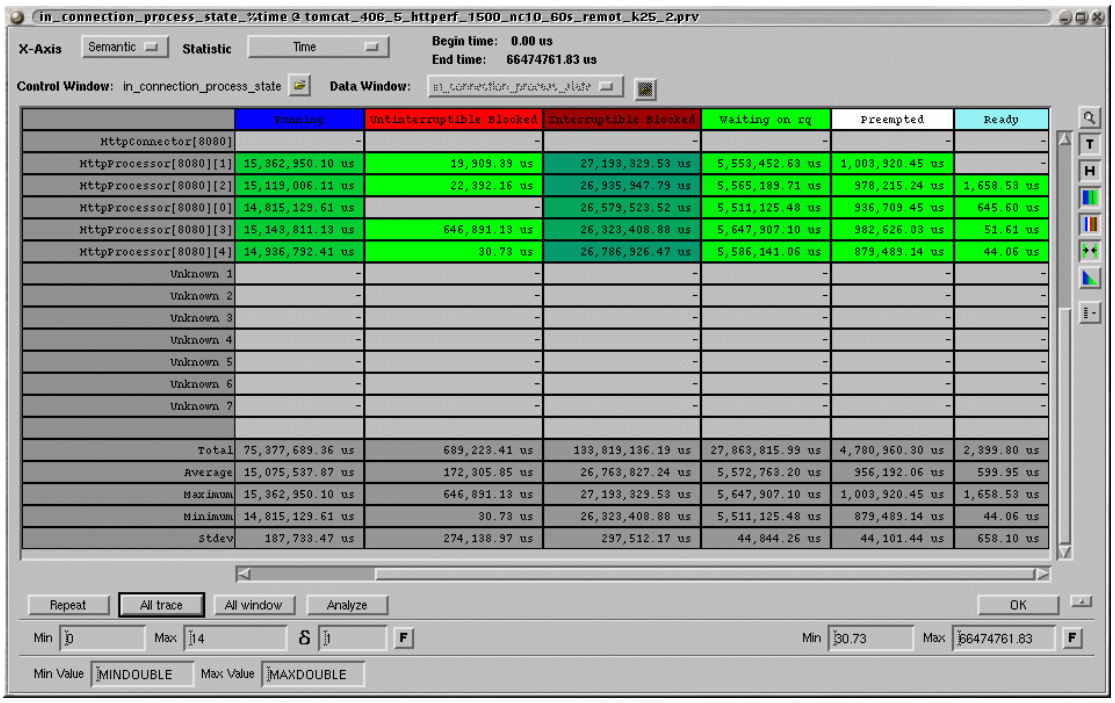
```
        Object : USER 8                     Click Time : 1742607 (us)

User Event at 1742612    Monitor Enter CounterClass@4F35708/4F9F990
User Event at 1742620    Monitor Entered CounterClass@4F35708/4F9F990
User Event at 1742630    Monitor Exit CounterClass@4F35708/4F9F990

 ==> Running    Duration : 5443
```

Figure 3.12. Sample Paraver graphical and textual visualizations

The analysis views apply statistical calculations to the trace file information and summarize the results as a table. These calculations can be done as a function of thread state values, punctual events and thread state values of one window in relation to thread state values (called categories) of another window. An example of a Paraver statistical analysis can be seen on Figure 3.13. On it, threads are represented on the vertical axis, categories are represented on the horizontal axis and each cell of the matrix represents the calculation result for the thread-category combination. The statistic to be calculated is defined by the user.


in_connection_process_state_%time @ tomcat_406_5_httperf_1500_nc10_60s_remot_k25_2.prv

X-Axis Semantic   Statistic Time   Begin time: 0.00 us   End time: 66474761.83 us

Control Window: in_connection_process_state   Data Window: in_connection_process_state

	Running	Untinterruptible Blocked	Interruptible Blocked	Waiting on rq	Preempted	Ready
HttpConnector[8080]	-	-	-	-	-	-
HttpProcessor[8080][1]	15,362,950.10 us	19,909.39 us	27,193,329.53 us	5,553,452.63 us	1,003,920.45 us	-
HttpProcessor[8080][2]	15,119,006.11 us	22,392.16 us	26,935,947.79 us	5,565,189.71 us	978,215.24 us	1,658.53 us
HttpProcessor[8080][0]	14,815,129.61 us	-	26,579,523.52 us	5,511,125.48 us	936,709.45 us	645.60 us
HttpProcessor[8080][3]	15,143,811.13 us	646,891.13 us	26,323,408.88 us	5,647,907.10 us	982,626.03 us	51.61 us
HttpProcessor[8080][4]	14,936,792.41 us	30.73 us	26,786,926.47 us	5,586,141.06 us	879,489.14 us	44.06 us
Unknown 1	-	-	-	-	-	-
Unknown 2	-	-	-	-	-	-
Unknown 3	-	-	-	-	-	-
Unknown 4	-	-	-	-	-	-
Unknown 5	-	-	-	-	-	-
Unknown 6	-	-	-	-	-	-
Unknown 7	-	-	-	-	-	-
Total	75,377,689.36 us	689,223.41 us	133,819,136.19 us	27,863,815.99 us	4,780,960.30 us	2,399.80 us
Average	15,075,537.87 us	172,305.85 us	26,763,827.24 us	5,572,763.20 us	956,192.06 us	599.95 us
Maximum	15,362,950.10 us	646,891.13 us	27,193,329.53 us	5,647,907.10 us	1,003,920.45 us	1,658.53 us
Minimum	14,815,129.61 us	30.73 us	26,323,408.88 us	5,511,125.48 us	879,489.14 us	44.06 us
Stdev	187,733.47 us	274,138.97 us	297,512.17 us	44,844.26 us	44,101.44 us	658.10 us

Min 0   Max 14   δ 1   Min 30.73   Max 66474761.83

Min Value MINDOUBLE   Max Value MAXDOUBLE

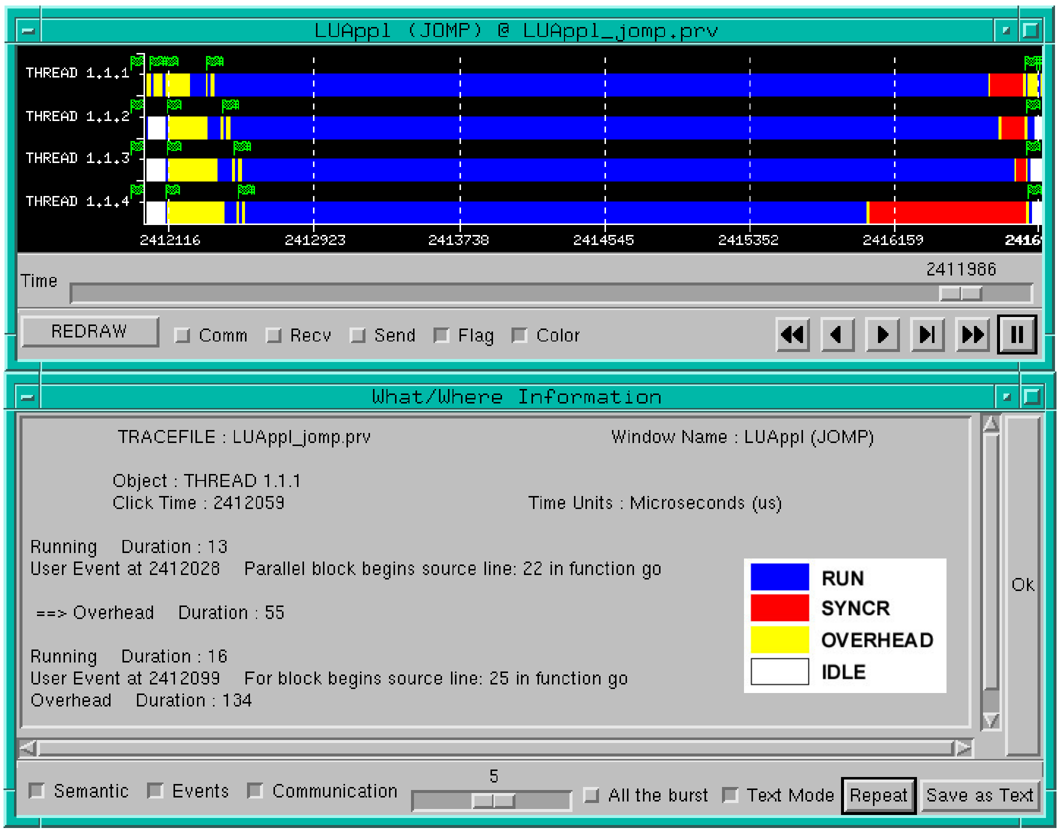
Figure 3.13. Sample Paraver statistical calculation

3.4 Application Analysis

Although many examples of using the proposed analysis framework for detecting performance bottlenecks of multithreaded Java applications will appear across this thesis, this section presents some guidelines of the analysis that can be performed with the analysis framework. This explanation has been structured around the analysis of two types of multithreaded Java applications: JOMP applications and Java application servers.

3.4.1 Analysis of JOMP Applications

The top part of Figure 3.14 shows a Paraver window displaying the behavior of one iteration of the LUJOMP application presented in Figure 3.11. The horizontal axis represents execution time in microseconds. The vertical axis shows the four JOMP threads that compose the team. Each thread evolves though a set of states, each one represented with a different color (as indicated with the legend). Flags appearing on top of each thread bar are used to visualize the events indicating the JOMP constructs. For instance, all the threads start executing the body of the parallel construct, and distribute themselves the work (`OVERHEAD` state, yellow color in the visualization) as indicated by the two for work-sharing directives. After determining the chunk of iterations, each thread executes them (`RUN` state, dark blue color in the visualization). Barrier synchronization happens at the end of second work-sharing construct (`SYNCH` state, red color in the visualization), which forces all the threads to wait. Notice that the `nowait` clause in the first work-sharing construct omits the implicit barrier synchronization.



```
TRACEFILE : LUAppl_jomp.prv                    Window Name : LUAppl (JOMP)

Object : THREAD 1.1.1
Click Time : 2412059                           Time Units : Microseconds (us)

Running    Duration : 13
User Event at 2412028    Parallel block begins source line: 22 in function go

 ==> Overhead    Duration : 55

Running    Duration : 16
User Event at 2412099    For block begins source line: 25 in function go
Overhead    Duration : 134
```

🟦	RUN
🟥	SYNCR
🟨	OVERHEAD
⬜	IDLE

Figure 3.14. Paraver visualization for one iteration of the LUAppl kernel (JOMP programming model level)

The bottom part of Figure 3.14 shows the textual information reported by Paraver when the user clicks on a specific point of the trace. Observe that, in addition to timing measurements and thread state, Paraver also relates the visualization with the original JOMP code.

The information obtained at the JOMP programming model level can be complemented with the information at the system level, allowing the correlation of the information at different levels in a way that cannot be accomplished with other tools. For example, the user can obtain information about the real processor use during the execution of JOMP constructs. In the top part of Figure 3.15, which shows the behavior at the JOMP programming model level of the LUJOMP application, the user can detect if a thread is in the `IDLE` state or in the `SYNCH` state, but it does not know if that thread is in fact running on a processor or blocked elsewhere on the system. In the bottom part of Figure 3.15, which shows the behavior at the system level of the LUJOMP application, the user discovers that when a thread is in the `IDLE` state, it is really in a loop checking for new work for be executed (`RUN` state) and if not found, yielding its processor (`READY` state). In the same way, when a thread is in the `SYNCH` state, it is really in a loop wait checking for the barrier opening (`RUN` state) and if do not, yielding its processor (`READY` state).

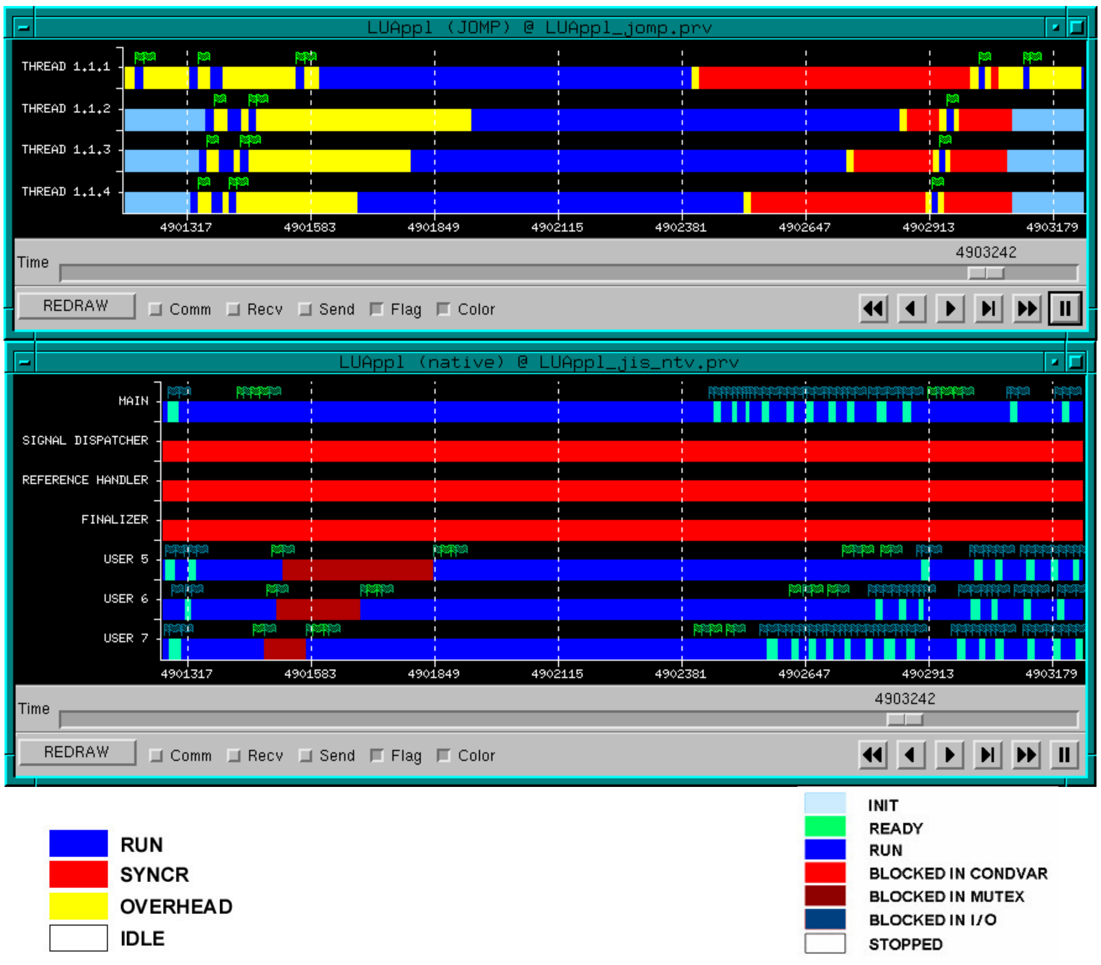


Figure 3.15. Paraver visualization for one iteration of the LUAppl kernel (JOMP programming model level + System level)

With the correlation of the information of the JOMP programming model level and the information of the system level, situations of monitor contention can be also discovered. The analysis of the top part of Figure 3.15 reveals that the overhead produced when distributing work among threads is higher than expected (`OVERHEAD` state). The analysis of the information at the system level, which is shown in the bottom part of Figure 3.15, exposes that a monitor contention situation in a JVM internal monitor is causing this overhead (threads are blocked in `BLOCKED_IN_MUTEX` state).

3.4.2 Analysis of Multithreaded Java Application Servers

This section presents two successful experiences where a detailed analysis using the proposed performance analysis framework has allowed the detection and correction of two performance degradation situations when executing the RUBiS benchmark with the Tomcat application server. The two analysis experiences demonstrate the benefit of disposing of correlated information about all the levels to perform a fine-grain analysis of server execution.

3.4.2.1 Analysis methodology

The analysis methodology is based in the well-know scientific method. The analysis starts when an observation that can represent a performance lost or a server malfunction is produced when doing typical server maintenance work (for example, when examining the server log files), or when performing a study of basic metrics looking for anomalous values or behaviors. These observations showing low performance or unexplained errors are the *Symptom* that something is going wrong in the server, and motivate an in-depth analysis of the server behavior.

When a *Symptom* of a server malfunction is detected, the analysis methodology indicates that a *Hypothesis* to explain this *Symptom* apparition have to be suggested, and using the performance analysis framework presented in this thesis, perform the necessary *Actions* to confirm or discard this *Hypothesis*. The result of the *Actions* can confirm the *Hypothesis*, discard it, or detect another *Symptom*. The methodology establishes to carry out the necessary *Actions* until the *Hypothesis* can be verified or discarded. In the first case, the cause of server anomalous behavior has been detected. In the second case, another *Hypothesis* must be suggested, and the verification process based on *Actions* must be restarted.

3.4.2.2 Case study 1

The first case study starts from an observation made when inspecting the Tomcat log files. Good Tomcat administrators should perform the observation of log files periodically in order to detect possible server malfunctions. When examining the RUBiS context log file of Tomcat, these error messages are found:

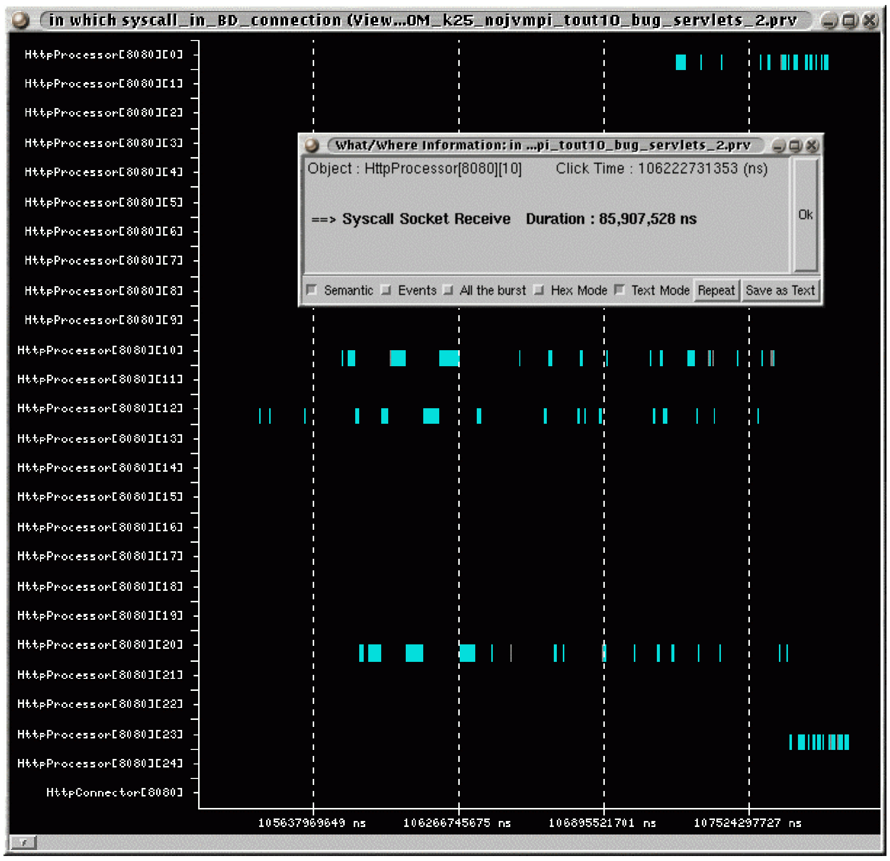
- ✓ `Servlet.service() for servlet BrowseCategories threw exception java.lang.NullPointerException at com.mysql.jdbc.ResultSet.first(ResultSet.java:2293)`
- ✓ `java.sql.SQLException: Operation not allowed after ResultSet closed`

The appearance of these error messages in the log file is a *Symptom* that something is going wrong, and motivates an in-depth analysis to determine the causes of this behavior. The proposed analysis methodology establishes the suggestion of a *Hypothesis* that explains the *Symptom* detected. Considering the messages shown before, the *Hypothesis* is that the problem is related with the database access.

At this point, it is required to take the necessary *Actions* to verify the *Hypothesis* made (using the performance analysis framework). In this case, correctness of database access has to be verified.

The first Action to verify the *Hypothesis* consists of analyzing which system calls are performed by HttpProcessors when they have acquired a database connection. This information is displayed in Figure 3.16 (horizontal axis is time and vertical axis identifies each thread), where each burst represents the execution of a system call when the corresponding HttpProcessor has acquired a database connection. As indicates the textual information in the figure, HttpProcessors get database information using socket receive calls. This *Symptom* corresponds to the expected behavior if managing correctly the database connections, thus more information about the database access is needed to verify the *Hypothesis*.

Then, the next *Action* taken is to analyze the file descriptors used by the system calls performed by HttpProcessors when they have acquired a database connection. This information is displayed in Figure 3.17, where each burst indicates the file descriptor used by the system call performed by the corresponding HttpProcessor when it has acquired a database connection. As indicates the textual information in the figure, several HttpProcessors are accessing the database using the same file descriptor (that is, using the same database connection). This is conceptually incorrect, and should not happen. This *Symptom* confirms the *Hypothesis* about a wrong access to database.

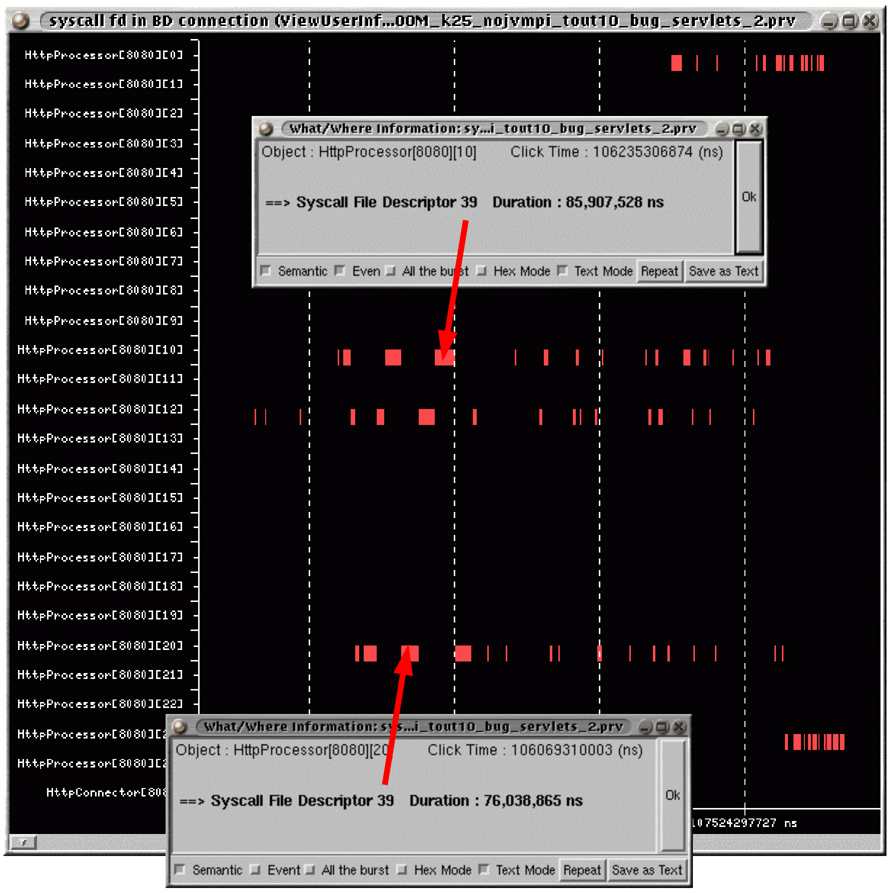


Object : HttpProcessor[8080][10]   Click Time : 106222731353 (ns)

==> Syscall Socket Receive   Duration : 85,907,528 ns

Figure 3.16. System calls performed by HttpProcessors when they have acquired a database connection

At this point, it must be determined why several HttpProcessor use the same file descriptor to access the database, so another *Hypothesis* that locates the problem in the RUBiS database connection management is suggested. The *Action* taken to verify this *Hypothesis* consists of inspecting the RUBiS servlets source code. This inspection reveals the following bug. Each kind of RUBiS servlet declares three class variables (`ServletPrinter sp`, `PreparedStatement stmt` and `Connection conn`). These class variables are shared by all the servlet instances, and this can provoke multiple race conditions. For example, it is possible that two HttpProcessors access the database using the same connection `conn`.



```
Object : HttpProcessor[8080][10]      Click Time : 106235306874 (ns)

==> Syscall File Descriptor 39   Duration : 85,907,528 ns
```

```
Object : HttpProcessor[8080][20      Click Time : 106069310003 (ns)

==> Syscall File Descriptor 39   Duration : 76,038,865 ns
```

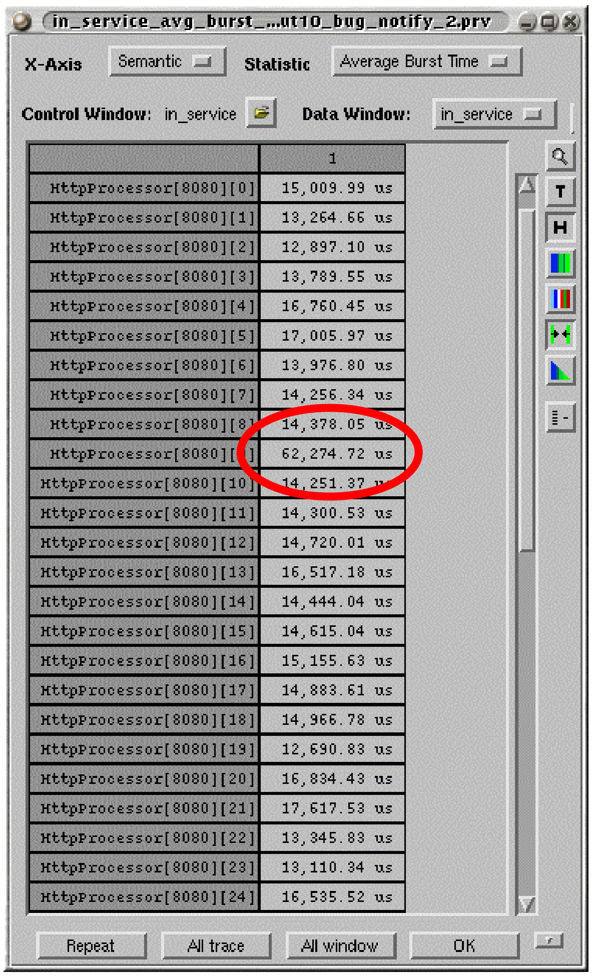
Figure 3.17. File descriptors used by the system calls performed by HttpProcessors when they have acquired a database connection

This problem can be avoided declaring these three class variables as local variables in the `doGet` method of the servlet, and pass them as parameters when needed.

3.4.2.3 Case study 2

A good practice when tuning an application server for performance is to make periodical studies of some basic metrics that indicate the performance of the application server. These metrics include for example the average service time per HttpProcessor, the overall throughput, the client requests arrivals rate, etc. The result of this basic analysis can encourage a more detailed study to determine the causes of an anomalous value in these metrics. For example, the second case study starts from an observation made when analyzing the average service time per HttpProcessor on server.

Figure 3.18 shows the average service time for each HttpProcessor, calculated using the performance analysis framework. In this figure there is one HttpProcessor with an average service time considerably higher than the others. This is a *Symptom* of an anomalous behavior of this HttpProcessor, and motivates an in-depth analysis to determine the causes of this behavior. First, the state distribution when the HttpProcessors are serving requests is analyzed. Figure 3.19 shows the percentage of time spent by the HttpProcessors on every state (`run`, `uninterruptible blocked`, `interruptible blocked`, `waiting in ready queue`, `preempted` and `ready`). This figure shows that the problematic HttpProcessor is most of the time in `interruptible blocked` state (about 92% of time) while the other HttpProcessors are blocked about the 65% of time.



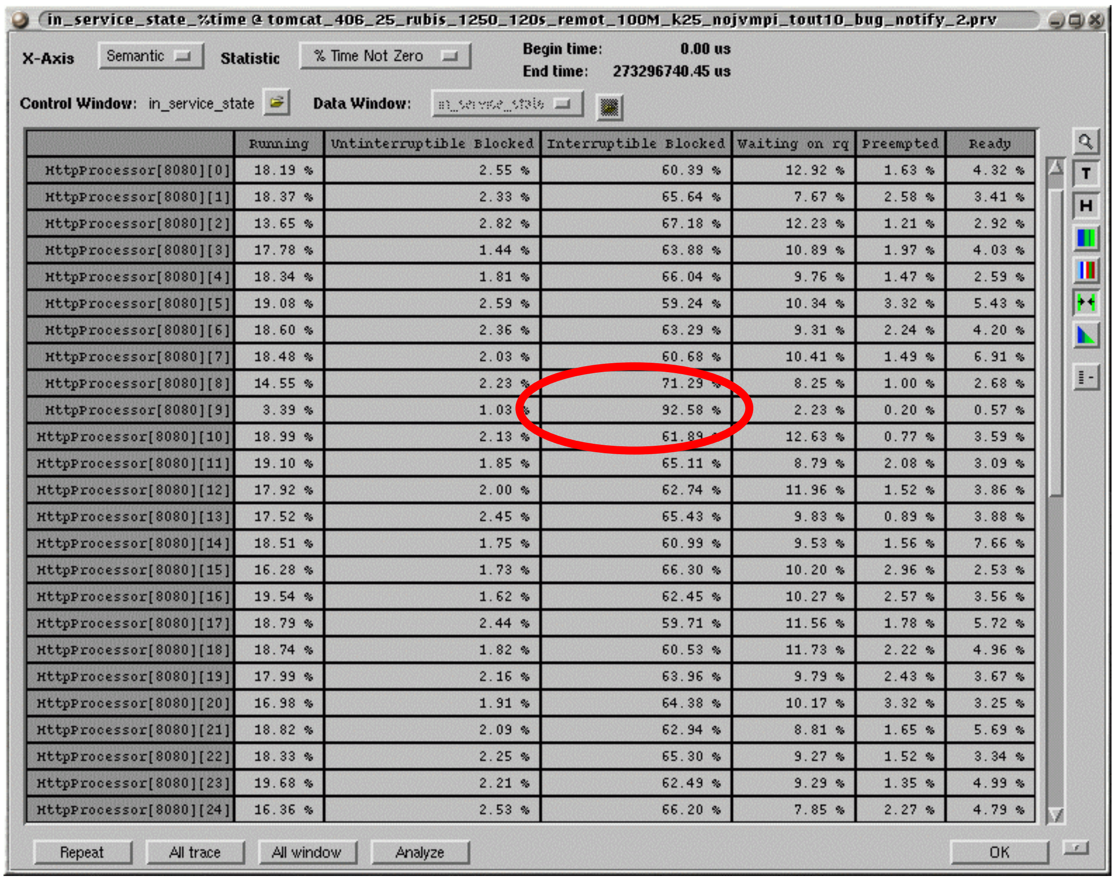
X-Axis Semantic Statistic Average Burst Time

Control Window: in_service Data Window: in_service

	1
HttpProcessor[8080][0]	15,009.99 us
HttpProcessor[8080][1]	13,264.66 us
HttpProcessor[8080][2]	12,897.10 us
HttpProcessor[8080][3]	13,789.55 us
HttpProcessor[8080][4]	16,760.45 us
HttpProcessor[8080][5]	17,005.97 us
HttpProcessor[8080][6]	13,976.80 us
HttpProcessor[8080][7]	14,256.34 us
HttpProcessor[8080][8]	14,378.05 us
HttpProcessor[8080][9]	62,274.72 us
HttpProcessor[8080][10]	14,251.37 us
HttpProcessor[8080][11]	14,300.53 us
HttpProcessor[8080][12]	14,720.01 us
HttpProcessor[8080][13]	16,517.18 us
HttpProcessor[8080][14]	14,444.04 us
HttpProcessor[8080][15]	14,615.04 us
HttpProcessor[8080][16]	15,155.63 us
HttpProcessor[8080][17]	14,883.61 us
HttpProcessor[8080][18]	14,966.78 us
HttpProcessor[8080][19]	12,690.83 us
HttpProcessor[8080][20]	16,834.43 us
HttpProcessor[8080][21]	17,617.53 us
HttpProcessor[8080][22]	13,345.83 us
HttpProcessor[8080][23]	13,110.34 us
HttpProcessor[8080][24]	16,535.52 us

Figure 3.18. Average service time per HttpProcessor

In order to explain this *Symptom*, the *Hypothesis* consists of assuming that the HttpProcessor could be blocked waiting response from the database. This *Hypothesis* is inferred because the database is a typical resource that can provoke long waits when working with application servers. To verify this *Hypothesis*, the *Action* taken is to analyze the system calls performed by HttpProcessors when serving requests. This analysis revealed that the problematic HttpProcessor is not blocked in any system call, which means that it is not blocked waiting response from database, but does it have at least an open connection with the database? To answer this question, the *Action* taken consists of analyzing when HttpProcessors acquire database connections. This analysis reports that the problematic HttpProcessor blocks before acquiring any database connection.


in_service_state_%time @ tomcat_406_25_rubis_1250_120s_remot_100M_k25_nojvmpi_tout10_bug_notify_2.prv

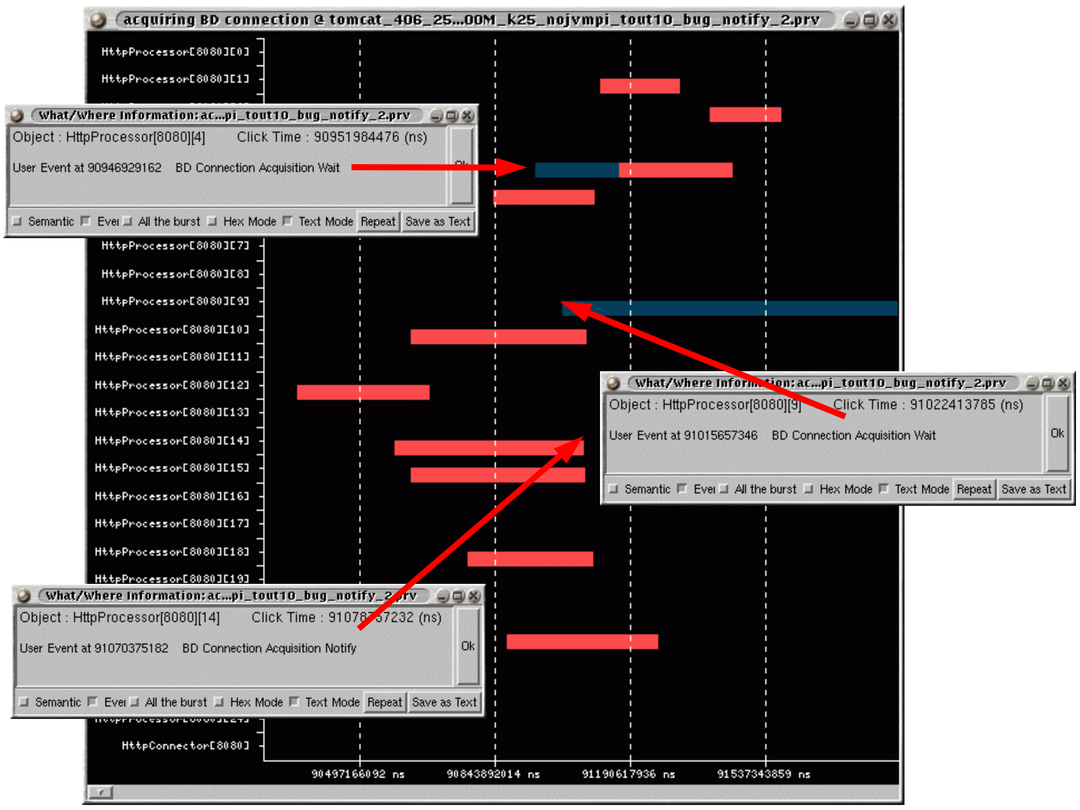
X-Axis: Semantic   Statistic: % Time Not Zero   Begin time: 0.00 us   End time: 273296740.45 us

Control Window: in_service_state   Data Window: in_service_state

	Running	Uninterruptible Blocked	Interruptible Blocked	Waiting on rq	Preempted	Ready
HttpProcessor[8080][0]	18.19 %	2.55 %	60.39 %	12.92 %	1.63 %	4.32 %
HttpProcessor[8080][1]	18.37 %	2.33 %	65.64 %	7.67 %	2.58 %	3.41 %
HttpProcessor[8080][2]	13.65 %	2.82 %	67.18 %	12.23 %	1.21 %	2.92 %
HttpProcessor[8080][3]	17.78 %	1.44 %	63.88 %	10.89 %	1.97 %	4.03 %
HttpProcessor[8080][4]	18.34 %	1.81 %	66.04 %	9.76 %	1.47 %	2.59 %
HttpProcessor[8080][5]	19.08 %	2.59 %	59.24 %	10.34 %	3.32 %	5.43 %
HttpProcessor[8080][6]	18.60 %	2.36 %	63.29 %	9.31 %	2.24 %	4.20 %
HttpProcessor[8080][7]	18.48 %	2.03 %	60.68 %	10.41 %	1.49 %	6.91 %
HttpProcessor[8080][8]	14.55 %	2.23 %	71.29 %	8.25 %	1.00 %	2.68 %
HttpProcessor[8080][9]	3.39 %	1.03 %	92.58 %	2.23 %	0.20 %	0.57 %
HttpProcessor[8080][10]	18.99 %	2.13 %	61.89 %	12.63 %	0.77 %	3.59 %
HttpProcessor[8080][11]	19.10 %	1.85 %	65.11 %	8.79 %	2.08 %	3.09 %
HttpProcessor[8080][12]	17.92 %	2.00 %	62.74 %	11.96 %	1.52 %	3.86 %
HttpProcessor[8080][13]	17.52 %	2.45 %	65.43 %	9.83 %	0.89 %	3.88 %
HttpProcessor[8080][14]	18.51 %	1.75 %	60.99 %	9.53 %	1.56 %	7.66 %
HttpProcessor[8080][15]	16.28 %	1.73 %	66.30 %	10.20 %	2.96 %	2.53 %
HttpProcessor[8080][16]	19.54 %	1.62 %	62.45 %	10.27 %	2.57 %	3.56 %
HttpProcessor[8080][17]	18.79 %	2.44 %	59.71 %	11.56 %	1.78 %	5.72 %
HttpProcessor[8080][18]	18.74 %	1.82 %	60.53 %	11.73 %	2.22 %	4.96 %
HttpProcessor[8080][19]	17.99 %	2.16 %	63.96 %	9.79 %	2.43 %	3.67 %
HttpProcessor[8080][20]	16.98 %	1.91 %	64.38 %	10.17 %	3.32 %	3.25 %
HttpProcessor[8080][21]	18.82 %	2.09 %	62.94 %	8.81 %	1.65 %	5.69 %
HttpProcessor[8080][22]	18.33 %	2.25 %	65.30 %	9.27 %	1.52 %	3.34 %
HttpProcessor[8080][23]	19.68 %	2.21 %	62.49 %	9.29 %	1.35 %	4.99 %
HttpProcessor[8080][24]	16.36 %	2.53 %	66.20 %	7.85 %	2.27 %	4.79 %

Figure 3.19. State distribution of HttpProcessors during service (in percentage)

With all this information it can be concluded that the first *Hypothesis* is wrong, that is, the problematic HttpProcessor is not waiting response from the database. Therefore, a new Hypothesis to explain why the problematic HttpProcessor is blocked most of the time is needed. Considering that, as commented before, the problematic HttpProcessor has not acquired any database connection yet, the new *Hypothesis* is that this HttpProcessor could have problems acquiring the database connection. To verify this *Hypothesis*, the performance analysis framework is used to display the database connections management, which is shown in Figure 3.20. Light color indicates the acquisition of a database connection and dark color indicates the wait for a free database connection. Notice that the problematic HttpProcessor (HttpProcessor 9 in the figure) is blocked waiting for a free database connection. This *Symptom* confirms the *Hypothesis* that there could be problems acquiring database connections. This figure also reveals the origin of the problem on the database connection management, because it can occur that a database connection is released, while there are some HttpProcessors waiting for a free database connection, but they are not notified. Notice that HttpProcessors 4 and 9 are blocked waiting for a free database connection. When HttpProcessor 14 releases its database connection, it notifies HttpProcessor 4 that can acquire this connection and continue its execution. Other HttpProcessors holding a database connection release it, but none of them notifies HttpProcessor 9.


acquiring BD connection @ tomcat_406_25...00M_k25_nojvmpi_tout10_bug_notify_2.prv

What/Where Information:ac...pi_tout10_bug_notify_2.prv
Object : HttpProcessor[8080][4]   Click Time : 90951984476 (ns)
User Event at 90946929162   BD Connection Acquisition Wait

What/Where Information:ac...pi_tout10_bug_notify_2.prv
Object : HttpProcessor[8080][9]   Click Time : 91022413785 (ns)
User Event at 91015657346   BD Connection Acquisition Wait

What/Where Information:ac...pi_tout10_bug_notify_2.prv
Object : HttpProcessor[8080][14]   Click Time : 91078757232 (ns)
User Event at 91070375182   BD Connection Acquisition Notify

Figure 3.20. Database connections acquisition process

Trying to explain this anomalous behavior, the *Hypothesis* supposes that a wrong database connection management at RUBiS is causing the problem. In order to verify this *Hypothesis*, the *Action* taken is to inspect the RUBiS servlets source code. This inspection reveals a bug. By default, in RUBiS one HttpProcessor only notifies a connection release if free database connection stack is empty. But consider the following situation:

There are N HttpProcessors that execute the same RUBiS servlet, which has a pool of M connections available with the database, where N is greater than M. This means that M HttpProcessors can acquire a database connection and the rest (N – M) HttpProcessors block waiting for a free database connection. Later, an HttpProcessor finishes executing the servlet and releases its database connection. The HttpProcessor puts the connection in the pool and, as the connection pool was empty, it notifies the connection release.

Due to this notification, a second HttpProcessor wakes up and tries to get a database connection. But before this second HttpProcessor can get the connection, a third HttpProcessor finishes executing the servlet and releases its database connection. The third HttpProcessor puts the connection in the pool and, as the connection pool was not empty (the second HttpProcessor has not got the connection yet), it does not notify the connection release. The second HttpProcessor finally acquires its database connection and the execution continues with a free connection in the pool, but with HttpProcessors still blocked waiting for free database connections.

This situation can be avoided if HttpProcessors notify to all HttpProcessors when they release a database connection.

3.5 Conclusions

This chapter has described the main contribution in the “Analysis and Visualization of Multithreaded Java Applications” work area of this thesis, which is the proposal of a performance analysis framework to perform a complete analysis of the Java applications behavior based on providing to the user detailed information about all levels involved in the application execution (operating system, JVM, application server and application), giving him the chance to construct his own metrics, oriented to the kind of analysis he wants to perform.

The performance analysis framework consists of two tools: an instrumentation tool, called JIS (Java Instrumentation Suite), and an analysis and visualization tool, called Paraver. When instrumenting a given application, JIS generates a trace in which the information collected from all levels has been correlated and merged. Later, the trace can be visualized and analyzed with Paraver (qualitatively and quantitatively) to identify the performance bottlenecks of the application.

JIS provides information from all levels involved in the application execution. From the system level, information about threads state and system calls (I/O, sockets, memory management and thread management) can be obtained. Several implementations have been performed depending on the underlying platform. A dynamic interposition mechanism that obtains information about the supporting threads layer (i.e. Pthreads library) without recompilation has been implemented for the SGI Irix platform. In the same way, a device driver that gets information from a patched Linux kernel has been developed for the Linux platform. JIS uses the JVMPI to obtain information from the JVM level. At this level of analysis, the user can obtain information about several Java abstractions like classes, objects, methods, threads and monitors, but JIS only obtains at this level the name of the Java threads and information from the different Java Monitors (when they are entered, exited or contended), due to the large overhead produced when using JVMPI. Information relative to services (i.e. servlets and EJB), requests, connections or transactions can be obtained from the application server level. Moreover, some extra information can be added to the final trace file by generating user events from the application code. Information at these levels can be inserted by hard-coding JNI calls to the instrumentation library on the server or the application source or by introducing them dynamically using Aspect programming techniques without source code recompilation.

As a special case of instrumentation at the application level, support for JOMP applications has been added to JIS. JOMP includes OpenMP-like extensions to specify parallelism in Java applications using a shared-memory programming paradigm. This instrumentation approach has been designed to provide a detailed analysis of the parallel behavior at the JOMP programming model level. At this level, the user is faced with parallel, work-sharing and synchronization constructs. The JOMP compiler has been modified to inject JNI calls to the instrumentation library during the code generation phase at specific points in the source code.

Experience in this thesis demonstrates the benefit of disposing of correlated information about all the levels involved in Java applications execution to perform a fine-grain analysis of their behavior. This thesis claims that a real performance improvement on multithreaded Java applications execution can only be achieved if performance bottlenecks at all levels can be identified.

The research performed in this work area has resulted in the following publications, including three international conferences, one international workshop and two national conferences:

- J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta. Tuning Dynamic Web Applications using Fine-Grain Analysis. 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'05), pp. 84-91, Lugano, Switzerland. February 9-11, 2005.

- D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. Complete Instrumentation Requirements for Performance Analysis of Web based Technologies. 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), pp. 166-175, Austin, Texas, USA. March 6-8, 2003.

- D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. An Instrumentation Tool for Threaded Java Application Servers. XIII Jornadas de Paralelismo, pp. 205-210, Lleida, Spain. September 9-11, 2002.

- J. Guitart, J. Torres, E. Ayguadé and J.M. Bull. Performance Analysis Tools for Parallel Java Applications on Shared-memory Systems. 30th International Conference on Supercomputing (ICPP'01), pp. 357-364, Valencia, Spain. September 3-7, 2001.

- J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Instrumentation Environment for Java Threaded Applications. XI Jornadas de Paralelismo, pp. 89-94. Granada, Spain, September 12-14, 2000.

- J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. 2nd Annual Workshop on Java for High Performance Computing (part of the 14th ACM International Conference on Supercomputing ICS'00), pp. 15-25, Santa Fe, New Mexico, USA. May 7, 2000.

CHAPTER 4
SELF-ADAPTIVE MULTITHREADED JAVA APPLICATIONS

4.1 Introduction

Multithreaded Java applications can be used in HPC environments, where applications can benefit from the Java multithreading support for performing parallel calculations, as well as in e-business environments, where Java application servers can take profit of Java multithreading facilities to handle concurrently a large number of requests.

However, the use of Java for HPC faces a number of problems that are currently subject of research. One of them is the performance degradation when multithreaded applications are executed in a multiprogrammed environment. The main issue that leads to this degradation is the lack of communication between the execution environment and the applications, which can cause these applications to make a naive use of threads, degrading their performance. In these situations, it is desirable that the execution environment provides information to the applications about their allocated resources, thus allowing the applications to adapt their behavior to the amount of resources offered by the execution environment by generating only the amount of parallelism that can be executed with the assigned processors. This capability of applications is known as malleability [53]. Therefore, improving the performance of multithreaded Java applications in HPC environments can be accomplished by designing and implementing malleable applications (i.e. self-adaptive applications).

Achieving good performance when using Java in e-business environments is a harder problem due to the large complexity of these environments. First, the workload of Internet sites is known to vary dynamically over multiple time scales, often in an unpredictable fashion, including flash crowds. This fact and the increasing load that Internet sites must support increase the performance demand on Java application servers that host the sites that must face situations with a large number of concurrent clients. Therefore, the scalability of these application servers has become a crucial issue in order to support the maximum number of concurrent clients in these situations.

Moreover, not all the web requests require the same computing capacity from the server. For example, requests for static web content (i.e. HTML files and images) are mainly I/O intensive. Requests for dynamic web content (i.e. Java Servlets and EJB) increase the computational demand on server, but often other resources (e.g. the database) become the bottleneck for performance. On the other side, in e-business applications, which are based on dynamic web content, all information that is confidential or has market value must be carefully protected when transmitted over the open Internet. Although providing these security capabilities does not introduce a new degree of complexity in web applications structure, it increases the computation time necessary to serve a connection remarkably, due to the use of cryptographic techniques, becoming a CPU-intensive workload.

Facing situations with a large number of concurrent clients and/or with a workload that demands high computational power (as for instance secure workloads) can lead a server to overload (i.e. the volume of requests for content at a site temporarily exceeds the capacity for serving them and renders the site unusable). During overload conditions, the response times may grow to unacceptable levels, and exhaustion of resources may cause the server to behave erratically or even crash causing denial of services. In e-commerce applications, which are heavily based on the use of security, such server behavior could translate to sizable revenue losses.

Therefore, overload prevention is a critical issue if good performance on Java application servers in e-business environments wants to be achieved. Overload prevention tries to have a system that remains operational in the presence of overload even when the incoming request rate is several times greater than system capacity, and at the same time is able to serve the maximum the number of requests during such overload, maintaining response times (i.e. Quality of Service (QoS)) within acceptable levels.

Additionally, in many web sites, especially in e-commerce, most of the applications are session-based. A session contains temporally and logically related request sequences from the same client. Session integrity is a critical metric in e-commerce. For an online retailer, the higher the number of sessions completed the higher the amount of revenue that is likely to be generated. The same statement cannot be made about the individual request completions. Sessions that are broken or delayed at some critical stages, like checkout and shipping, could mean loss of revenue to the web site. Sessions have distinguishable features from individual requests that complicate the overload control. For example, admission control on per request basis may lead to a large number of broken or incomplete sessions when the system is overloaded.

Application servers overload can be prevented by designing mechanisms that allow the servers to adapt their behavior to the available resources (i.e. becoming self-adaptive applications) limiting the number of accepted requests to those that can be served without degrading their QoS while prioritizing important requests. However, the design of a successful overload prevention strategy must be preceded by a complete characterization of the application server scalability. This characterization allows determining which factors are the bottlenecks for application server performance that must be considered in the overload prevention strategy.

Nevertheless, characterizing application server scalability is something more complex than measuring the application server performance with different number of clients and determining the load that overloads the server. A complete characterization must also supply the causes of this overload, giving to the server administrator the chance and the information to improve the server scalability by avoiding its overload. For this reason, this characterization requires of 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 and consider all the levels involved in the execution of web applications if they want to provide meaningful 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: the scalability relative to the resources. The analysis of the application server behavior will assist with hints to answer the question about how would affect to the application server scalability the addition of more resources. If the analysis reveals that some resource is being a bottleneck for the application server performance, this encourages the addition of new resources of this type in order to improve server scalability. On the other side, if a resource that is not being a bottleneck for the application server performance is upgraded, the added resources are wasted because the scalability is not improved and the causes of server performance degradation remain unresolved.

The first contribution of this thesis in the "Self-Adaptive Multithreaded Java Applications" work area is a complete characterization of the scalability of Java application servers when running secure dynamic web applications divided in two parts. The first one consists of measuring Tomcat vertical scalability (i.e. adding more processors) when using SSL determining the impact of adding more processors on server overload. The second one involves a detailed analysis of the server behavior using the performance analysis framework presented in Chapter 3, in order to determine the causes of the server overload when running with different number of processors.

The conclusions derived from this analysis demonstrate the convenience of incorporating to the application server (and give hints for its implementation) an overload control mechanism that is the second contribution of this thesis in the "Self-Adaptive Multithreaded Java Applications" work area. The overload control mechanism is based on SSL connections differentiation and admission control. SSL connections differentiation is accomplished by proposing a possible extension of the Java Secure Sockets Extension (JSSE) package to distinguish SSL connections depending on if the connection will reuse an existing SSL connection on the server or not. This differentiation can be very useful in order to design intelligent overload control policies on server, given the big difference existing on the computational demand of new SSL connections versus resumed SSL connections. Based on this SSL connections differentiation, a session-based adaptive admission control mechanism for the Tomcat application server is implemented. This mechanism allows the server to avoid throughput degradation and response time increments occurred on server saturation. The server differentiates full SSL connections from resumed SSL connections limiting the acceptation of full SSL connections to the maximum number acceptable with the available resources without overloading, while accepting all the resumed SSL connections. Moreover, the admission control mechanism maximizes the number of sessions completed successfully, allowing to e-commerce sites based on SSL to increase the number of transactions completed, thus generating higher benefit.

4.2 Self-Adaptive Multithreaded Java Applications in HPC Environments

As commented before, self-adaptive multithreaded Java applications in HPC environments can be obtained by designing and implementing malleable applications, that is, applications able to adapt their behavior to the amount of resources offered by the execution environment by generating only the amount of parallelism that can be executed with the assigned processors. Next section describes how this capability can be achieved for JOMP applications used in this thesis as a particular case of multithreaded Java applications in HPC environments.

4.2.1 Self-Adaptive JOMP Applications

By default, a JOMP application executes with as many threads as indicated in one of the arguments of the interpreter command line (`–Djomp.threads`). Nevertheless, the JOMP application can change its concurrency level (the amount of parallelism that will be generated in the next parallel region) inside any sequential region invoking the `setNumThreads()` method from the JOMP runtime library.

4.3 Self-Adaptive Multithreaded Java Applications Servers in e-Business Environments

4.3.1 Scalability Characterization of Multithreaded Java Application Servers in Secure Environments

4.3.1.1 Scalability characterization methodology

The scalability of an application server is defined as the ability to maintain a site availability, reliability, and performance as the amount of simultaneous web traffic, or load, hitting the application server increases [78].

Given this definition, the scalability of an application server can be represented measuring the performance of the application server while the load increases. With this representation, the load that overloads the server can be detected. An application server is overloaded when it is unable to maintain the site availability, reliability, and performance (i.e. the server does not scale). As derived from the definition, when the server is overloaded, the performance is degraded (lower throughput and higher response time) and the number of client requests refused is increased.

At this point, two questions should appear to the reader (and of course, to the application server administrator). First, the load that overloads the server has been detected, but why is this load causing 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 when the server is overloaded? In order to answer this question, this thesis proposes to analyze the application server behavior using the performance analysis framework presented in Chapter 3, which considers all levels involved in the application server execution, allowing a fine-grain analysis of dynamic web applications.

Second, the application server scalability with given resources has been measured, but how would affect to the application server scalability the addition of more resources? This adds a new dimension to the 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 to no changes to the architecture of the system. Vertical scalability increases the performance (in theory) and the manageability of the system, but decreases the reliability and availability (single failure is more likely to lead to system failure). This thesis considers this kind of scalability relative to the resources.

Horizontal scalability (also called scaling out) is achieved by adding new application servers to the system, increasing its complexity. Horizontal scalability increases the reliability, the availability and the performance (depends on load balancing), but decreases the manageability (there are more elements in the system).

The analysis of the application server behavior will assist with hints to answer the question about how would affect to the application server scalability the addition of more resources. If some resource is being a bottleneck for the application server performance, this encourages the addition of new resources of this type (vertical scaling), the measurement of the scalability with this new configuration and the analysis of the application server behavior with the performance analysis framework to determine the improvement on the server scalability and the new causes of server overload.

On the other side, if a resource that is not being a bottleneck for the application server performance is upgraded, it can be verified with the performance analysis framework that scalability is not improved and the causes of server performance degradation remain unresolved. This observation justifies why with vertical scalability performance is improved only in theory, depending if the added resource is a bottleneck for server performance or not. This observation also motivates the analysis of the application server behavior in order to detect the causes of overload before adding new resources.

4.3.1.2 Scalability characterization of the Tomcat server

This section presents the scalability characterization of Tomcat application server when running the RUBiS benchmark using SSL. The characterization is divided in two parts. The first part is an evaluation of the vertical scalability of the server when running with different number of processors, determining the impact of adding more processors on server overload (can the server support more clients before overloading?). The second part consists of a detailed analysis of the server behavior using the performance analysis framework, in order to determine the causes of the server overload when running with different number of processors.

4.3.1.2.1 Vertical scalability of the Tomcat server

Figure 4.1 shows the Tomcat scalability when running with different number of processors, representing the server throughput as a function of the number of concurrent 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 4.1 shows the number of clients that overload 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 overloading, so the maximum achieved throughput is higher.

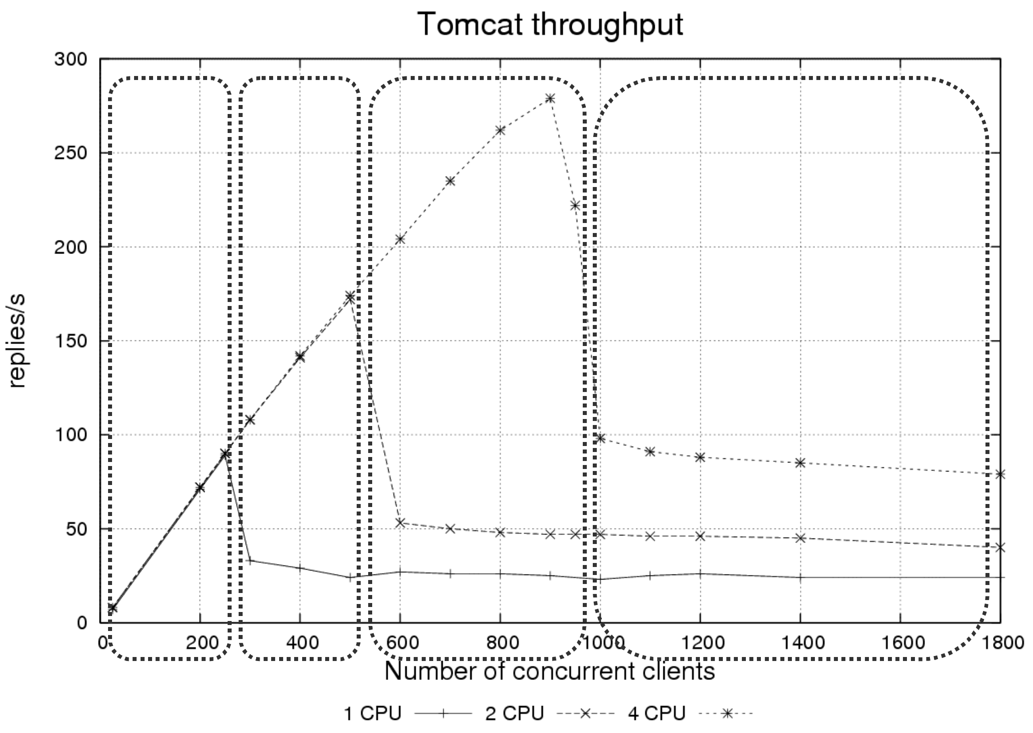


Figure 4.1. Tomcat scalability with different number of processors

Notice also that the same throughput can be achieved, as shown in Figure 2.3, 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.

Table 4.1. Number of clients that overload the server and maximum achieved throughput before overloading

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

When the number of clients that overload the server has been achieved, the server throughput degrades to approximately the 30% of the maximum achievable throughput, as shown in Table 4.2. This table shows the average throughput obtained when the server is overloaded when running with one, two and four processors. Notice that, although the throughput obtained has been degraded in all cases when the server has reached an overloaded state, running with more processors improves the throughput (duplicating the number of processors, the throughput almost duplicates too).

Table 4.2. Average server throughput when it is overloaded

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

4.3.1.2.2 Scalability analysis of the Tomcat server

In order to perform a detailed analysis of the server, four different loads have been selected: 200, 400, 800 and 1400 clients, each one corresponding to one of the zones observed in Figure 4.1. These zones group the loads with similar behavior of the server. The analysis is conducted using the performance analysis framework described in Chapter 3.

The analysis methodology consists of comparing the server behavior when it is overloaded (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). A series of metrics representing the server behavior are calculated, determining which of them are affected when increasing the number of clients. From these metrics, an in-depth analysis is performed looking for the causes of their dependence of server load.

The first metric calculated, using the performance analysis framework, is the average time spent by the server processing a persistent client connection, distinguishing the time devoted to each phase of the connection (persistent connection phases have been described in Section 2.3.3) when running with different number of processors. This information is displayed in Figure 4.2. As shown in this figure, running with more processors decreases the average time required to process a connection. Notice that when the server is overloaded, the average time required to handle a connection increases considerably. Going into detail on the connection phases, the time spent in 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 the major part of the time when processing a connection.

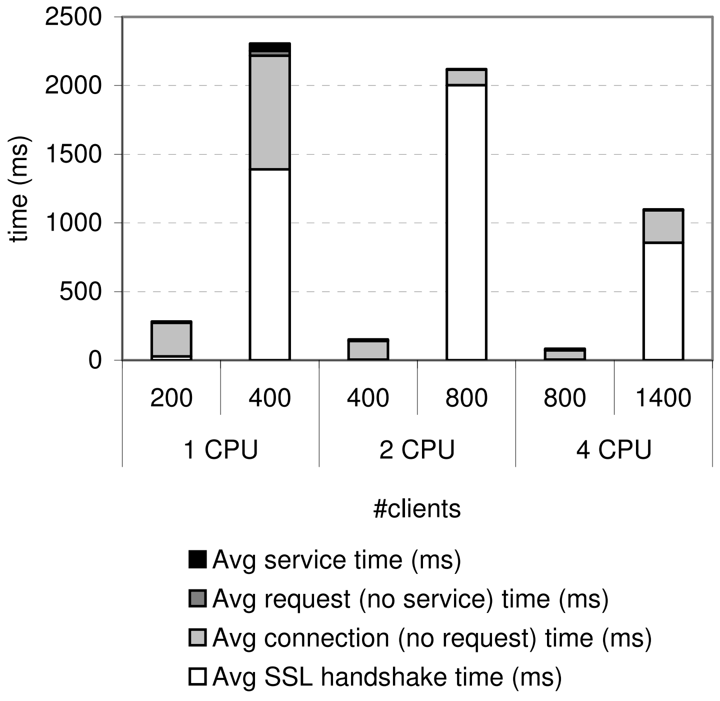


Figure 4.2. Average time spent by the server processing a persistent client connection

To determine the causes of the large increment of the time spent in the *SSL handshake* phase of the connection, the next step consists of calculating the percentage of connections that perform a resumed SSL handshake (reusing the SSL Session ID) versus the percentage of connections that perform a full SSL handshake when running with different number of processors. This information is shown in Figure 4.3. Notice that when running with one processor and with 200 clients, the 97% of SSL handshakes can reuse the SSL connection, but with 400 clients, only the 27% can reuse it. The rest must negotiate the full SSL handshake, overloading 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%).

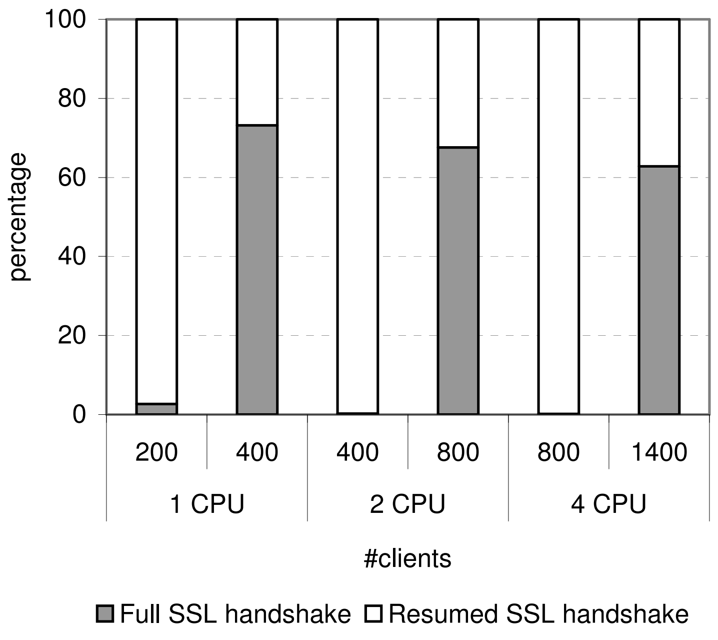


Figure 4.3. Incoming SSL connections classification depending on SSL handshake type performed

The analysis performed has determined that when running with any number of processors the server overloads 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, why does this occur from 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 the client has been configured 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 that the initiation of a new client requires the establishment of a new SSL connection, and therefore the negotiation of a full SSL handshake.

Therefore, if the server is loaded and it cannot handle the incoming requests before the client timeouts expire, this provokes the arrival of a large amount of new client connections that need the negotiation of a full SSL handshake, provoking the server performance degradation. This asseveration is supported with the information displayed in Figure 4.4. This figure shows the number of clients timeouts occurred when running with different number of processors. Notice that from 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 4.1 reveals that this given number of clients matches with the load that overloads the server.

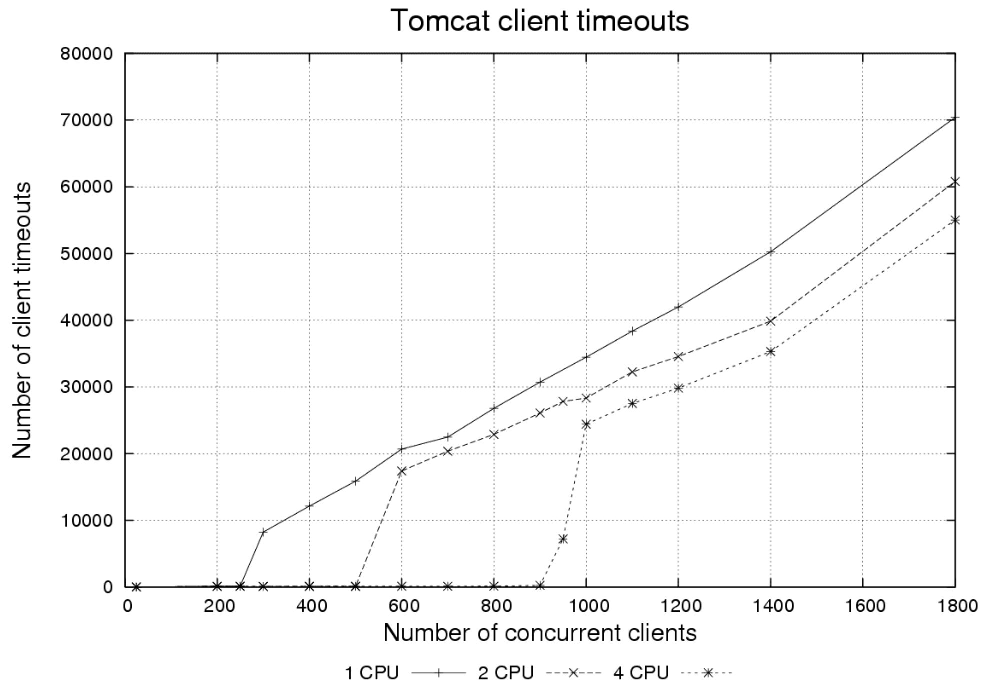


Figure 4.4. Client timeouts with different number of processors

In order to evaluate the effect on server of the large amount of full SSL handshakes, the performance analysis framework is used to calculate the state of HttpProcessors when they are in the *SSL handshake* phase of the connection, which is shown in Figure 4.5. 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` state) or waiting for a free processor to become available to execute (`Ready` state). When the server is not overloaded, HttpProcessors spend most of their time in `Run` state. But when the server is running with one processor and overloads (400 clients) HttpProcessors spend the 47% of their time in `Ready` state. This fact confirms that the server cannot handle all the incoming full SSL handshakes with only one processor.

It is expected that when the server is overloaded and running with two or four processors, the HttpProcessors spend most part of their time of `Ready` state (waiting for a free processor to execute), in the same way as when running with one processor.

But Figure 4.5 shows that, although the time spent on `Ready` state has increased when the server is running with two processors and overloads, the HttpProcessors spend the 70% of their time in `Blocked Synch` state (blocked waiting for the synchronization with other HttpProcessors in a monitor). This kind of contention can be produced due to the saturation of the available processors on multiprocessor systems, as occurred in this case. When running with four processors, the time spent in `Ready` state and `Blocked Synch` state is also increased.

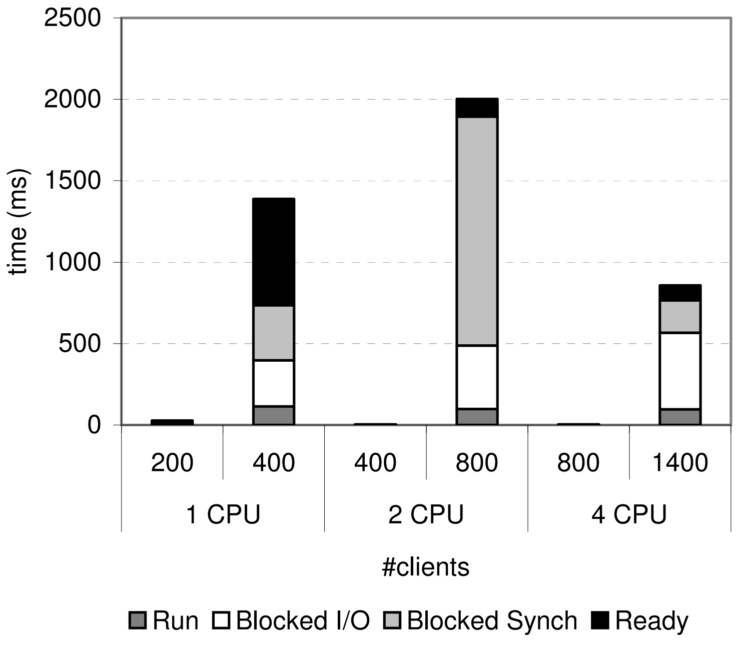


Figure 4.5. State of HttpProcessors when they are in the 'SSL handshake' phase of a connection

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

The analysis performed allows concluding that the processor is a bottleneck for Tomcat performance and scalability when running dynamic web applications in a secure environment. The analysis has demonstrated that running with more processors makes the server able to handle more clients before overloading, and even when the server has reached an overloaded state, better throughput can be obtained if running with more processors.

The results of the analysis performed in this section demonstrate the convenience of incorporating to the Tomcat server some kind of overload control mechanism to avoid the throughput degradation produced due to the massive arrival of new SSL connections. The server could differentiate new SSL connections from resumed SSL connections limiting the acceptation of new SSL connections to the maximum number acceptable without overloading, while accepting all the resumed SSL connections to maximize the number of client sessions successfully completed.

4.3.2 Session-Based Adaptive Overload Control for Multithreaded Java Application Servers in Secure Environments

4.3.2.1 SSL connections differentiation

As mentioned in Section 2.3.5.2, there is no way in JSSE packages to consult if an incoming SSL connection provides a reusable SSL session ID until the handshake is fully completed. This thesis proposes the extension of the JSSE package to allow applications to differentiate new SSL connections from resumed SSL connections prior the handshaking has started.

This new feature can be useful in many scenarios. For example, a connection scheduling policy based on prioritizing the resumed SSL connections (that is, the short connections) will result in a reduction of the average response time, as described in previous works with static web content using the SRPT scheduling [46][80]. Moreover, prioritizing the resumed SSL connections will increase the probability for a client to complete a session, maximizing the number of sessions completed successfully. The importance of this metric in e-commerce environments has been already commented. Remember that the higher the number of sessions completed the higher the amount of revenue that is likely to be generated. In addition, a server could limit the number of new SSL connections that it accepts, in order to avoid throughput degradation produced if server overloads.

In order to evaluate the advantages of being able to differentiate new SSL connections from resumed SSL connections and the convenience of adding this functionality to the standard JSSE package, this thesis includes the implementation of an experimental mechanism that allows this differentiation prior to the handshake negotiation. Performed measurements denote that this mechanism does not suppose significant additional cost. The mechanism works at system level and it is based on examining the contents of the first TCP segment received on the server after the connection establishment.

After a new connection is established between the server and a client, the SSL protocol starts a handshake negotiation. The protocol begins with the client sending a SSL ClientHello message (see the RFC 2246 for more details) to the server. This message can include a SSL session ID from a previous connection if the SSL session wants to be reused. This message is sent in the first TCP segment that the client sends to the server. The implemented mechanism checks the value of this SSL message field to decide if the connection is a resumed SSL connection or a new one instead.

The mechanism operation begins when the Tomcat server accepts a new incoming connection, and a socket structure is created to represent the connection in the operating system as well as in the JVM. After establishing the connection but prior to the handshake negotiation, the Tomcat server requests to the mechanism the classification of this SSL connection, using a JNI native library that is loaded into the JVM process. The library translates the Java request into a new native system call implemented in the Linux kernel using a Linux kernel module.

The implementation of the system call calculates a hash function from the parameters provided by the Tomcat server (local and remote IP address and TCP port) which produces a socket hash code that makes possible to find the socket inside of a connection established socket hash table. When the system `struct sock` that represents the socket is located and in consequence all the received TCP segments for that socket after the connection establishment, the first one of the TCP segments is interpreted as a SSL ClientHello message. If this message contains a SSL session ID with value 0, it can be concluded that the connection tries to establish a new SSL session. If a non-zero SSL session ID is found instead, the connection tries to resume a previous SSL session. The value of this SSL message field is returned by the system call to the JNI native library that, in turn, returns it to the Tomcat server. With this result, the server can decide, for instance, to apply an admission control algorithm in order to decide if the connection should be accepted or rejected. A brief diagram of the mechanism operation described above can be found in Figure 4.6.

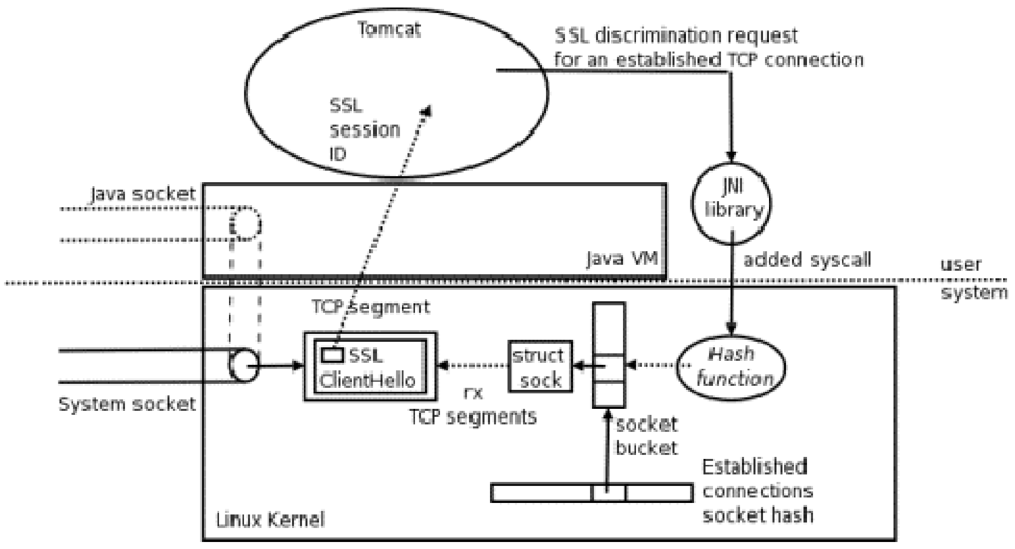


Figure 4.6. SSL connections differentiation mechanism

4.3.2.2 SSL admission control

In order to prevent server overload in secure environments, this thesis proposes to incorporate to the Tomcat server a session-oriented adaptive mechanism that performs admission control based on SSL connections differentiation. This mechanism has been developed with two objectives. First, to prioritize the acceptation of client connections that resume an existing SSL session, in order to maximize the number of sessions successfully completed. Second, to limit the massive arrival of new SSL connections to the maximum number acceptable by the server before overloading, depending on the available resources.

To prioritize the resumed SSL connections, the admission control mechanism accepts all the connections that supply a valid SSL session ID. The required verification to differentiate resumed SSL connections from new SSL connections is performed with the mechanism described in Section 4.3.2.1.

To avoid the server throughput degradation and maintain acceptable response times, the admission control mechanism must to avoid the server overload. By keeping the maximum amount of load just below the system capacity, overload is prevented and peak throughput is achieved. For servers running secure web applications, the system capacity depends on the available processors, as it has been demonstrated in Section 4.3.1, due to the large computational demand of this kind of applications. Therefore, if the server can use more processors, it can accept more SSL connections without overloading.

The admission control mechanism calculates periodically, introducing an adaptive behavior, the maximum number of new SSL connections that can be accepted without overloading the server. This maximum depends on the available processors for the server and the computational demand required by the accepted resumed SSL connections. The calculation of this demand is based on the number of accepted resumed SSL connections and the typical computational demand of one of these connections.

After calculating the computational demand required by the accepted resumed SSL connections and with information relative to the available processors for the server, the admission control mechanism can calculate the remaining computational capacity for attending new SSL connections. The admission control mechanism will only accept the maximum number of new SSL connections that do not overload the server (they can be served with the available computational capacity). The rest of new SSL connections arriving at the server will be refused.

Notice that if the number of resumed SSL connections increases, the server has to decrease the number of new SSL connections it accepts, in order to avoid server overload with the available processors and vice versa, if the number of resumed SSL connections decreases, the server can increase the number of new SSL connections that it accepts.

Notice that this constitutes an interesting starting point to develop autonomic computing strategies on the server in a bi-directional fashion. First, the server can restrict the number of new SSL connections it accepts to adapt its behavior to the available resources (i.e. processors) in order to prevent server overload. Second, the server can inform about its resource requirements to a global manager (which will distribute all the available resources among the existing servers following a given policy) depending on the rate of incoming connections (new SSL connections and resumed SSL connections) requesting for service.

4.3.2.3 Evaluation

This section presents the evaluation results when comparing the performance of the Tomcat server with the overload control mechanism with respect to the original Tomcat. These results are obtained using a slightly different methodology with respect to Section 4.3.1. This section calculated server scalability by measuring the server throughput as a function of the number of concurrent clients. The number of concurrent clients that a server can handle without overloading is an important reference in current web sites, because if a site is able to support more concurrent clients, more benefit is likely to be generated for the site.

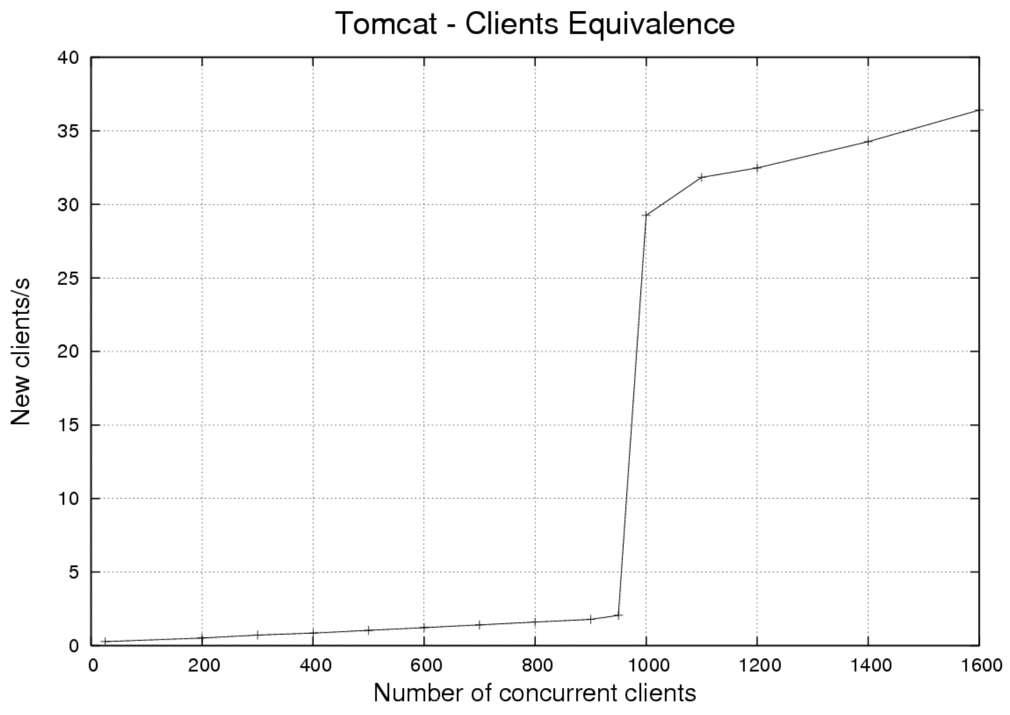


Figure 4.7. Equivalence between new clients per second and concurrent clients

However, the scalability characterization has revealed that when the server overloads, a small increment in the number of concurrent clients produces great throughput degradation. This effect can be explained with the information in Figure 4.7. This figure shows the number of new clients per second initiating a session with the server as a function on the number of concurrent clients. Notice that, when the number of concurrent clients that overloads the server has been achieved, the number of new clients per second initiating a session with the server increases exponentially. As these new clients must negotiate a full SSL handshake, this causes the server throughput degradation.

In order to avoid this behavior, and make the overload process of the server more progressive, the performance measurements of the server for the experiments in this section are relative to the number of new clients per second initiating a session with the server instead of being relative to the number of concurrent clients.

Measuring in this way makes easier to analyze the server behavior when overloads and the proposal and implementation of overload control mechanisms.

4.3.2.3.1 Original Tomcat server

Figure 4.8 shows the Tomcat throughput as a function of the number of new clients per second initiating a session with the server when running with different number of processors. 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. Notice that running with more processors allows the server to handle more clients before overloading, so the maximum achieved throughput is higher. When the number of clients that overload the server has been achieved, the server throughput degrades until approximately the 20% of the maximum achievable throughput while the number of clients increases.

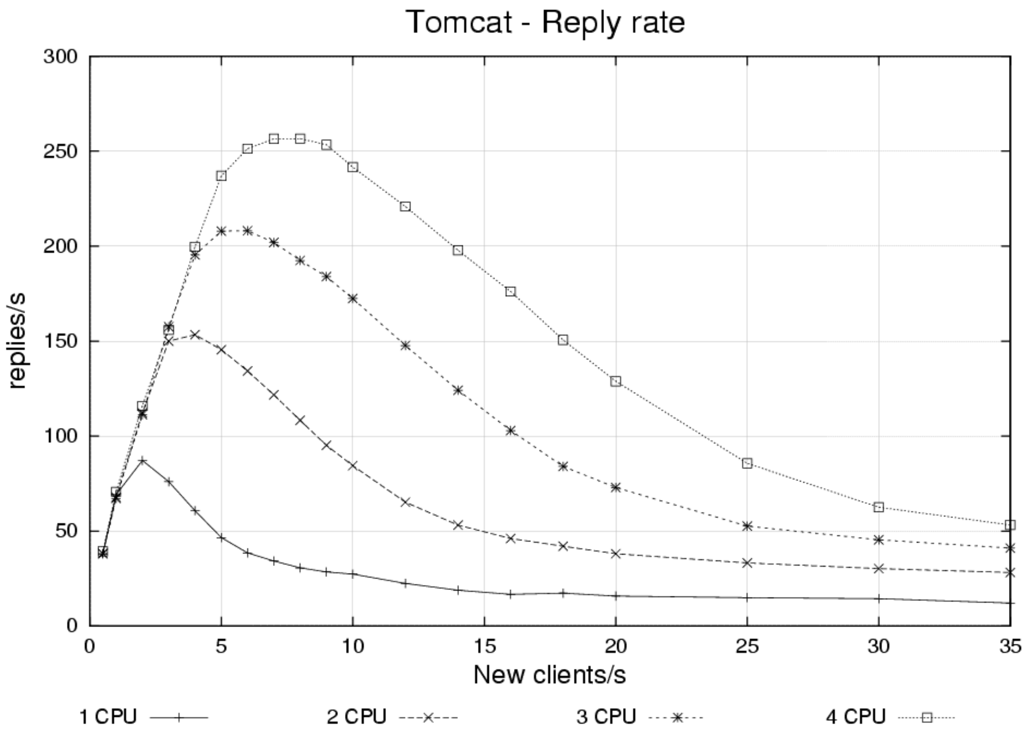


Figure 4.8. Original Tomcat throughput with different number of processors

As well as degrading the server throughput, the server overload also affects to the server response time, as shown in Figure 4.8. This figure shows the server average response time as a function of the number of new clients per second initiating a session with the server when running with different number of processors. Notice that when the server is overloaded the response time increases (especially when running with one processor) while the number of clients increases.

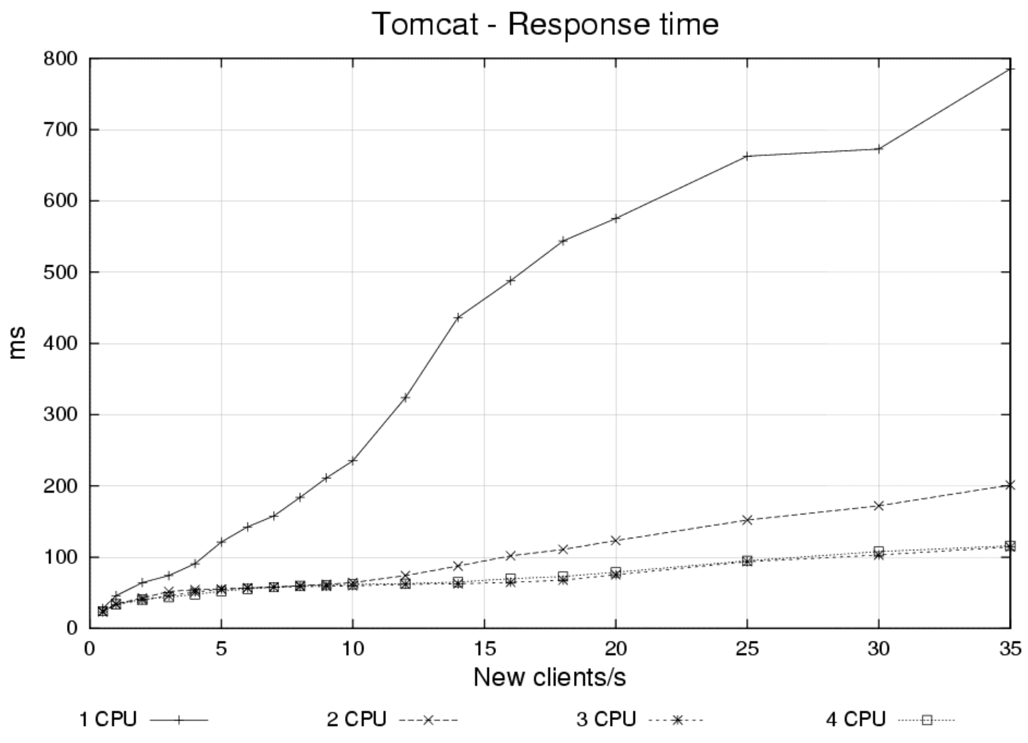


Figure 4.9. Original Tomcat response time with different number of processors

Server overload has another undesirable effect, especially in e-commerce environments where session completion is a key factor. As shown in Figure 4.10, which shows the number of sessions completed successfully when running with different number of processors, only a few sessions can finalize completely when the server is overloaded. Consider the large revenue lost that this fact can provoke for example in an online store, where only a few clients can finalize the acquisition of a product.

The cause of this large performance degradation on server overload has been analyzed in Section 4.3.1.2.2. This section concludes that the server throughput degrades 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. This circumstance is produced when the server is overloaded and it cannot handle the incoming requests before the client timeouts expire. In this case, clients with expired timeouts are discarded and new ones are initiated, provoking the arrival of a large amount of new client connections that negotiate of a full SSL handshake, provoking server performance degradation.

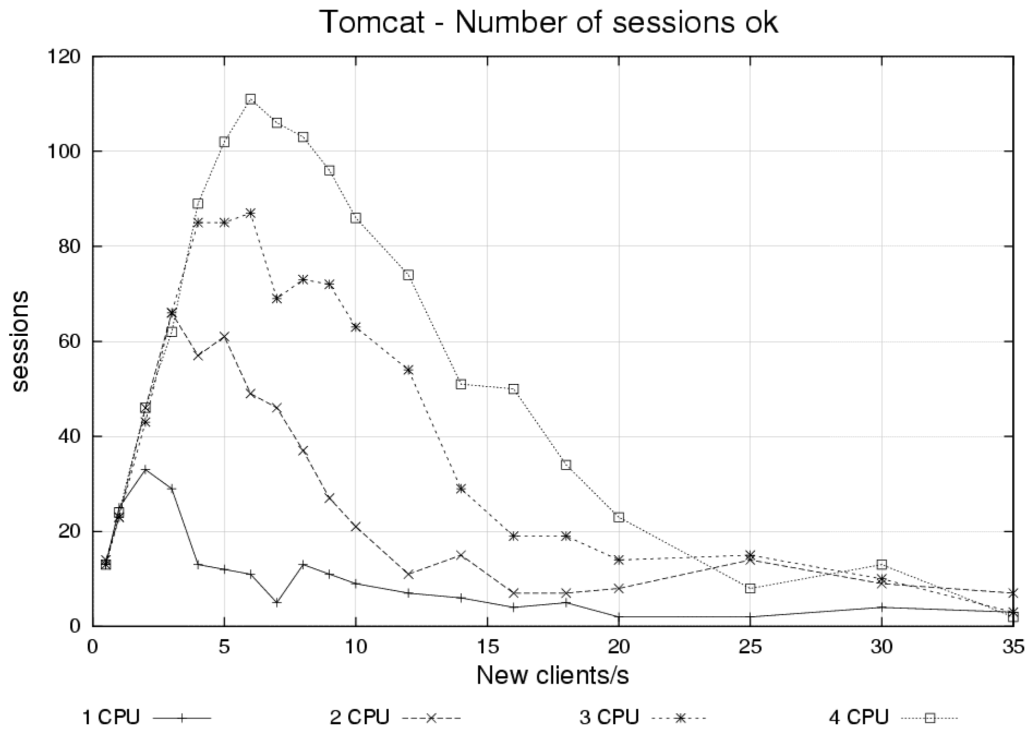


Figure 4.10. Completed sessions by original Tomcat with different number of processors

Considering the described behavior, it makes sense to apply an admission control mechanism in order to improve server performance in the following way. First, to filter the massive arrival of client connections that need to negotiate a full SSL handshake that will overload the server, avoiding the server throughput degradation and maintaining a good quality of service (good response time) for already connected clients. Second, to prioritize the acceptation of client connections that resume an existing SSL session, in order to maximize the number of sessions successfully completed.

4.3.2.3.2 Self-adaptive Tomcat server

Figure 4.11 shows the Tomcat throughput as a function of the number of new clients per second initiating a session with the server when running with different number of processors. 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. Until this point, the server with admission control behaves in the same way than the original server. However, when the number of clients that would overload the server has been achieved, the admission control mechanism can avoid the throughput degradation, maintaining it in the maximum achievable throughput, as shown in Figure 4.11. Notice that running with more processors allows the server to handle more clients, so the maximum achieved throughput is higher.

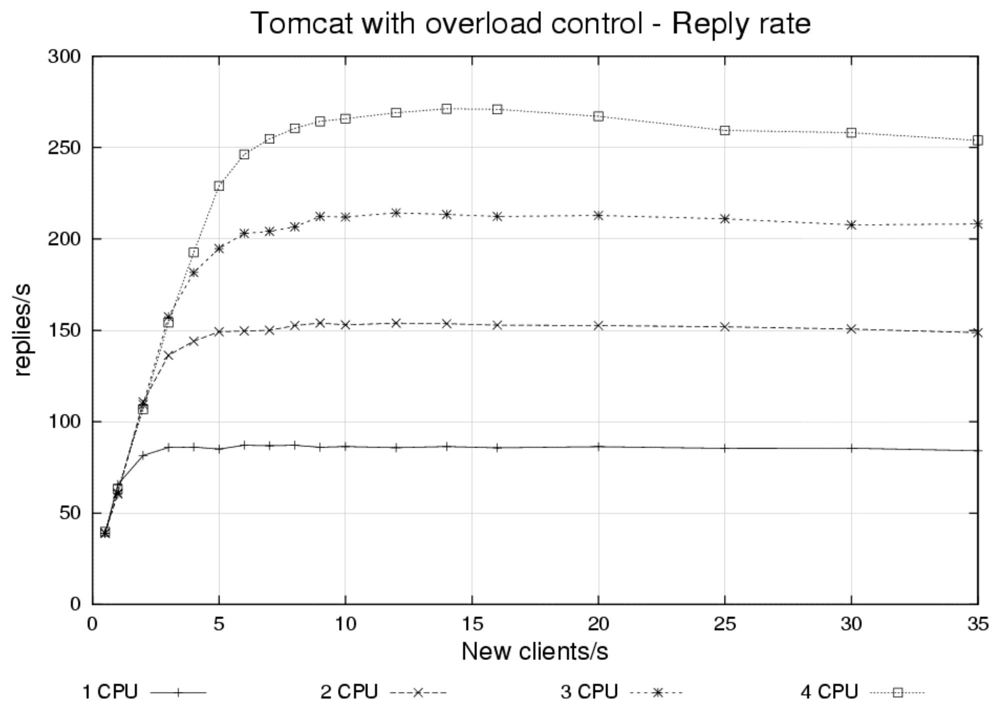


Figure 4.11. Tomcat with admission control throughput with different number of processors

The admission control mechanism on Tomcat allows also maintaining the response time in levels that guarantee a good quality of service to the clients, even when the number of clients that would overload the server has been achieved, as shown in Figure 4.12. This figure shows the server average response time as a function of the number of new clients per second initiating a session with the server when running with different number of processors.

Finally, the admission control mechanism has also a beneficial effect for session-based clients. As shown in Figure 4.13, which shows the number of sessions finalized successfully when running with different number of processors, the number of sessions that can finalize completely does not decrease, even when the number of clients that would overload the server has been achieved.

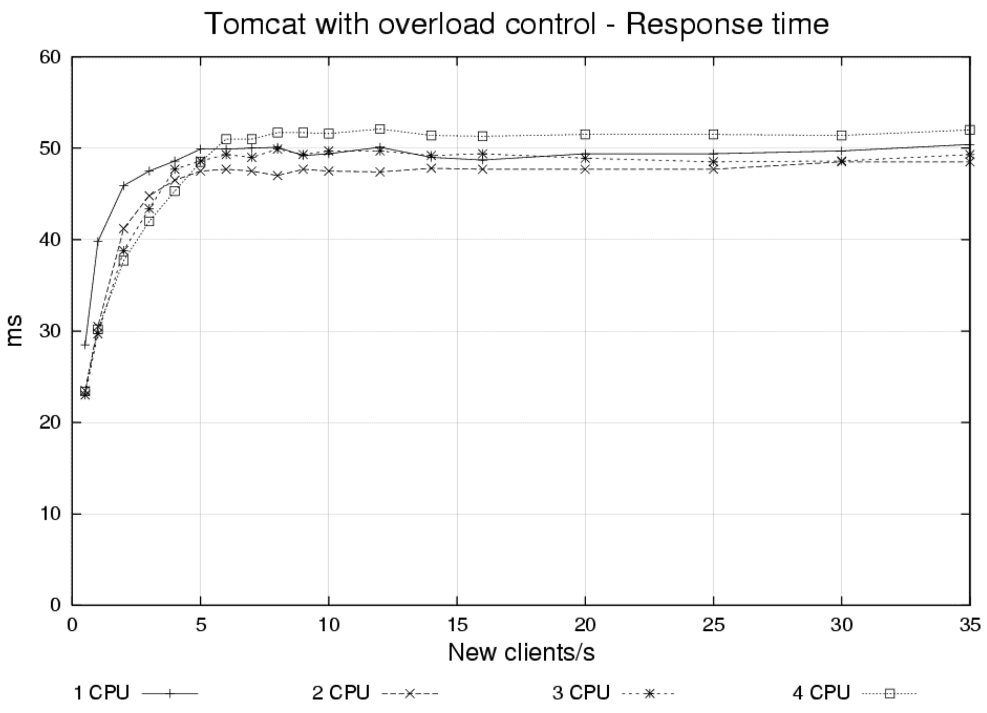


Figure 4.12. Tomcat with admission control response time with different number of processors

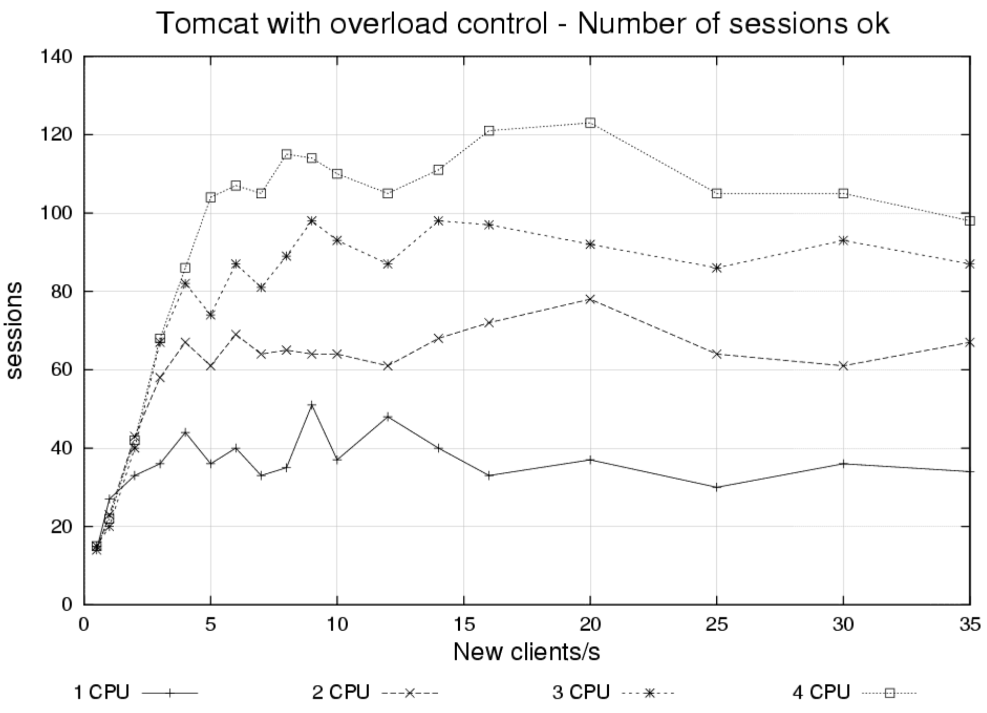


Figure 4.13. Sessions completed by Tomcat with admission control with different number of processors

4.4 Conclusions

The “Self-Adaptive Multithreaded Java Applications” work area described in this chapter, demonstrate the benefit of implementing self-adaptive multithreaded Java applications in order to achieve good performance as in HPC environments as in e-business environments. Self-adaptive applications are those applications that can adapt their behavior to the amount of resources allocated to them.

This chapter has presented two contributions towards achieving self-adaptive applications. The first contribution is a complete characterization of the scalability of Java application servers when executing secure dynamic web applications. This characterization is divided in two parts:

The first part has consisted of measuring Tomcat vertical scalability (i.e. adding more processors) when using SSL and analyzing the effect of this addition on server scalability. The results have confirmed that running with more processors makes the server able to handle more clients before overloading and even when the server has reached an overloaded state, better throughput can be obtained if running with more processors. The second part has involved an analysis of the causes of server overload when running with different number of processors using the performance analysis framework proposed in Chapter 3 of this thesis. The analysis has revealed that the processor is a bottleneck for Tomcat performance on secure environments (the massive arrival of new SSL connections demands a computational power that the system is unable to supply and the performance is degraded) and could make sense to upgrade the system adding more processors to improve the server scalability. The analysis results also have demonstrated the convenience of incorporating to the Tomcat server some kind of overload control mechanism to avoid the throughput degradation produced due to the massive arrival of new SSL connections that the analysis has detected.

Based on the conclusions extracted from this analysis, the second contribution is the implementation of a session-based adaptive overload control mechanism based on SSL connections differentiation and admission control. SSL connections differentiation has been accomplished using a possible extension of the JSSE package in order to allow distinguishing resumed SSL connections (that reuse an existing SSL session on server) from new SSL connections. This feature has been used to implement a session-based adaptive admission control mechanism that has been incorporated to the Tomcat server. This admission control mechanism differentiates new SSL connections from resumed SSL connections limiting the acceptation of new SSL connections to the maximum number acceptable with the available resources without overloading the server, while accepting all the resumed SSL connections in order to maximize the number of sessions completed successfully, allowing to e-commerce sites based on SSL to increase the number of transactions completed.

The experimental results demonstrate that the proposed mechanism prevents the overload of Java application servers in secure environments. It maintains response time in levels that guarantee good QoS and avoids completely throughput degradation (throughput degrades until approximately the 20% of the maximum achievable throughput when server overloads), while maximizes the number of sessions completed successfully (which is a very important metric on e-commerce environments). These results confirm that security must be considered as an important issue that can heavily affect the scalability and performance of Java application servers.

However, although the admission control mechanisms maintain the QoS of admitted requests even during overloads, a significant fraction of the requests may be turned away during extreme overloads. In such a scenario, an increase in the effective application server capacity is necessary to reduce the request drop rate. This can be accomplished by allowing the cooperation of the application servers with the execution environment in the resource management. In this way, when the application server is overloaded, it can request additional resources to the execution environment, which decides the resources distribution among application servers in the system using policies that can include business indicators. At this point, the application server can use the admission control mechanism developed in this thesis to adapt its incoming workload to the assigned capacity. The description of this cooperation for resource provisioning is presented in Chapter 5.

The research performed in this work area has resulted in the following publications, including two international conferences and one national conference:

- J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. Session-Based Adaptive Overload Control for Secure Dynamic Web Applications. 34th International Conference on Supercomputing (ICPP’05), pp. 341-349, Oslo, Norway. June 14-17, 2005.

- J. Guitart, V. Beltran, D. Carrera, J. Torres and E. Ayguadé. Characterizing Secure Dynamic Web Applications Scalability. 19th International Parallel and Distributed Symposium (IPDPS'05), Denver, Colorado, USA. April 4-8, 2005.

- V. Beltran, J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta. Performance Impact of Using SSL on Dynamic Web Applications. XV Jornadas de Paralelismo, pp. 471-476, Almeria, Spain. September 15-17, 2004.

CHAPTER 5 RESOURCE PROVISIONING FOR MULTITHREADED JAVA APPLICATIONS

5.1 Introduction

In the way towards achieving good performance when running multithreaded Java applications either in HPC environments or in e-business environments, this thesis has demonstrated in Chapter 4 that having self-adaptive multithreaded Java applications can be very useful to achieve this objective.

However, the maximum effectiveness for preventing applications performance degradation in parallel environments is obtained when fitting the self-adaptation of the applications to the available resources within a global strategy in which the execution environment and the applications cooperate to manage the resources efficiently.

For example, besides of having self-adaptive Java applications in HPC environments, performance degradation of multithreaded Java applications in these environments can only be avoided if overcoming the following limitations. First, the Java runtime environment does not allow applications to have control on the number of kernel threads where Java threads map and to suggest about the scheduling of these kernel threads. Second, the Java runtime environment does not inform the applications about the dynamic status of the underlying system so that the self-adaptive applications cannot adapt their execution to these characteristics. Finally, the large number of migrations of the processes allocated to an application occurred, due to scheduling polices that do not consider multithreaded Java applications as an allocation unit.

The same applies to Java application servers in e-business environments. In this case, although the admission control mechanisms used to implement self-adaptive applications in this scenario can maintain the quality of service of admitted requests even during overloads, a significant fraction of the requests may be turned away during extreme overloads. In such a scenario, an increase in the effective server capacity is necessary to reduce the request drop rate. In fact, although several techniques have been proposed to face with overload, such as admission control, request scheduling, service differentiation, service degradation or resource management, last work in this area has demonstrated that the most effective way to handle overload considers a combination of these techniques [140].

For these reasons, this thesis contributes in the “Resource Provisioning for Multithreaded Java Applications” work area with the proposal of mechanisms to allow the cooperation between the applications and the execution environment in order to improve the performance by managing resources efficiently in the framework of Java applications, including the modifications that are required in the Java execution environment to allow this cooperation. The cooperation is implemented by establishing a bi-directional communication path between the applications and the underlying system. On one side, the applications request to the execution environment the number of processors they need. On the other side, the execution environment can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications, which are self-adaptive, can adapt their behavior to the amount of resources allocated to them.

In order to accomplish this resource provisioning strategy in HPC environments, this thesis shows that the services supplied by the Java native underlying threads library, in particular the services to inform the library about the concurrency level of the application, are not enough to support the cooperation between the applications and the execution environment, because this uni-directional communication does not allow the application to adapt its execution to the available resources. In order to address the problem, the thesis proposes to execute the self-adaptive multithreaded Java applications on top of JNE (Java Nanos Environment built around the Nano-threads environment [101]). JNE is a research platform that provides mechanisms to establish a bi-directional communication path between the Java applications and the execution environment, thus allowing applications to collaborate in the thread management.

In e-business environments, the resource provisioning strategy is accomplished using an overload control approach for self-adaptive Java application servers running secure e-commerce applications that brings together admission control based on SSL connections differentiation and dynamic provisioning of platform resources in order to adapt to changing workloads avoiding the QoS degradation. Dynamic provisioning enables additional resources to be allocated to an application on demand to handle workload increases, while the admission control mechanisms maintain the QoS of admitted requests by turning away excess requests and preferentially serving preferred clients (to maximize the generated revenue) while additional resources are being provisioned.

The overload control approach is based on a global resource manager responsible of distributing periodically the available resources (i.e. processors) among web applications in a hosting platform applying a given policy (which can consider e-business indicators). This resource manager and the applications cooperate to manage the resources using a bi-directional communication. On one side, the applications request to the resource manager the number of processors needed to handle their incoming load avoiding the QoS degradation. On the other side, the resource manager can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications, which are self-adaptive, apply the admission control mechanism described in Chapter 4 to adapt their incoming workload to the assigned capacity by limiting the number of admitted requests accepting only those that can be served with the allocated processors without degrading their QoS.

5.2 Resource Provisioning for Multithreaded Java Applications in HPC Environments

5.2.1 Motivating Example

In order to demonstrate the performance degradation of multithreaded Java applications when running in multiprogrammed HPC environments, this section presents a simple experiment based on LUAppl, a LU reduction kernel over a two-dimensional matrix of double-precision elements taken from [111] that uses a matrix of 1000x1000 elements. The experiment consists of a set of executions of LUAppl running with different number of Java threads and active kernel threads (with a processor assigned to them). Table 5.1 shows the average execution time on a SGI Origin 2000 architecture [129] with MIPS R10000 processors at 250 MHz running SGI Irix JVM version Sun Java Classic 1.2.2. The first and second rows show that when the number of Java threads matches the number of active kernel threads, the application benefits from running with more threads. However, if the number of active kernel threads provided to support the execution does not match, as shown in the third row, the performance is degraded. In this case the execution environment (mainly the resource manager in the kernel) is providing only three active kernel threads, probably because either there are no more processors available to satisfy the application requirements, or the execution environment is unable to determine the concurrency level of the application. In the first case, this situation results in an execution time worse than the one achieved if the application would have known that only three processors were available and would have adapted its behavior to simply generate three Java threads (like in the first row). In the second case, this situation results in an execution time worse than the one achieved if the execution environment would have known the concurrency level of the application and would have provided four active kernel threads (like in the second row).

Table 5.1. LUAppl performance degradation

Java threads	**Active kernel threads**	**Execution time (in seconds)**
3	**3**	39.7
4	**4**	34.3
4	**3**	44.1

This thesis considers two different ways of approaching the problem in the Java context. The first one simply uses one of the services supplied by the Java native underlying threads library to inform the library about the concurrency level of the application. In the second one, Java applications are executed on top of JNE (Java Nanos Environment built around the Nano-threads environment [101]). JNE provides the mechanisms to establish a bi-directional communication path between the application and the underlying system.

5.2.2 Concurrency Level

The experimental environment is based on the SGI Irix JVM, which like many others (Linux, Solaris, Alpha, IBM, etc.) implements the native threads model using the Pthreads [121] library. Thus, one Java thread maps directly into one pthread, and the Pthreads library is responsible for scheduling these pthreads over the kernel threads offered by the operating system.

Version Sun Java Classic 1.2.2 of SGI Irix JVM does not inform the underlying threads layer about the desired concurrency level of the application. By default, the threads library adjusts the level of concurrency itself as the application runs using metrics that include the number of user context switches and CPU bandwidth. In order to provide the library with a more accurate hint about the concurrency level of the application, the programmer could invoke, at appropriate points in the application, the `pthread_setconcurrency(level)` service of the Pthreads library. The argument `level` is used by Pthreads to compute the ideal number of kernel threads required to schedule the available Java threads.

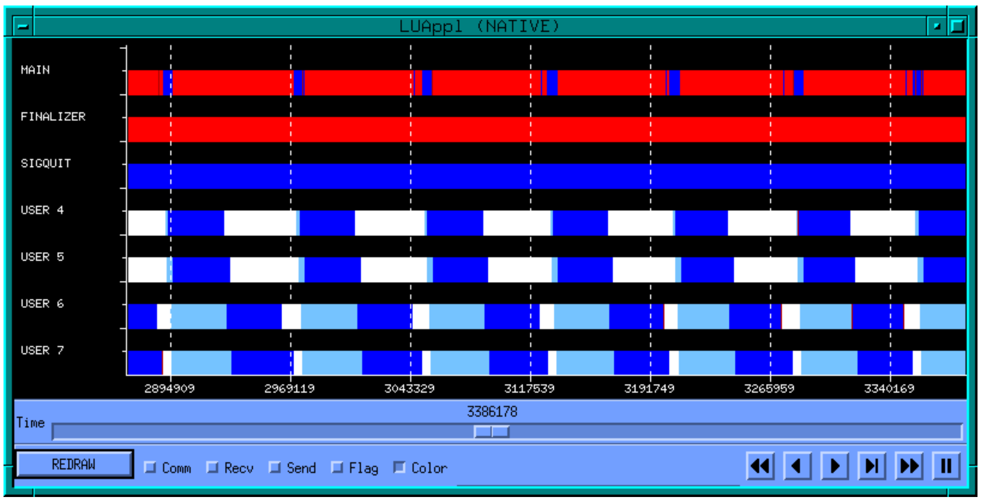


Figure 5.1. Paraver window showing LUAppl behavior without setting the concurrency level

Previous experimentation has revealed that informing to the threads library about the concurrency level of the application may have an important incidence on performance. The experimented improvements range from 23% to 58% when executing applications that create threads with a short lifetime. Threads are so short that the threads library is unable to estimate the concurrency level of the application and provide it with the appropriate number of kernel threads. This effect can be appreciated in Figure 5.1, which shows a Paraver window displaying the execution of a LUAppl that creates four Java threads but does not set the concurrency level. Notice that, although four threads are created, only two threads provide parallelism. When a hint of the concurrency level is provided by the application, the underlying threads library is capable of immediately providing the necessary kernel threads as shown in Figure 5.2.

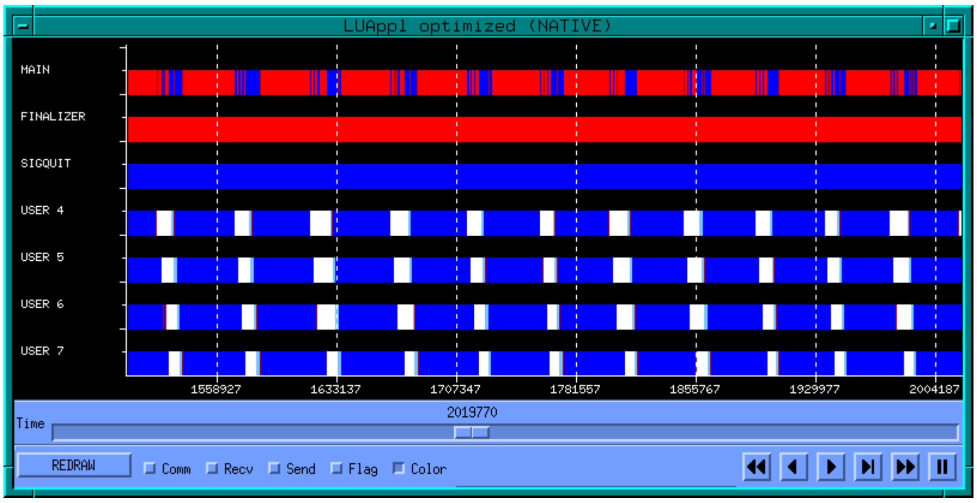


Figure 5.2. Paraver window showing LUAppl behavior setting the concurrency level

For those parallel Java applications that create threads with a long lifetime, such as the Java Grande benchmarks used in this thesis, informing about the concurrency level has less impact on performance. For this kind of applications, the threads library has time enough to estimate and adjust the number of kernel threads required during the thread lifetime. However, the time required to estimate the concurrency level of the application is not negligible and may approach the order of hundreds of milliseconds (even a few seconds depending of the application). Therefore, providing this hint is beneficial in any case.

In summary, this approach only solves one of the problems when running multithreaded Java applications in multiprogrammed HPC environments. Applications can inform to the execution environment about their processor requirements. However, other problems remain open. For instance, this approach does not allow applications to decide about the scheduling of kernel threads. Besides, the execution environment cannot inform each application about the number of processors actually assigned to it. As a consequence, applications cannot react and adapt their behavior to the decisions taken by the underlying system. If informed, applications would be able to restrict themselves in terms of parallelism generation, thus avoiding unnecessary overheads, balancing executions and exploiting available resources.

Newer versions of the SGI Irix JVM (from Sun Java 1.3) incorporate this approach and set the concurrency level to the maximum number of processors available in the system, obtaining performance gains similar to the ones obtained with the concurrency level approach (having also the same problems).

5.2.3 Java Nanos Environment (JNE)

The Java Nanos Environment (JNE) is a research platform that provides additional mechanisms to improve the communication between multithreaded Java applications and the underlying execution environment, thus allowing applications to collaborate in the thread management. JNE is able to solve many of the drawbacks appeared when running multithreaded Java applications in multiprogrammed HPC environments. First, JNE allows to the applications to have control on how Java threads maps onto kernel threads, specifying the number of processors on which the application wants to run at any moment. Second, JNE allows to the applications to decide about the scheduling of kernel threads, specifying one of the policies supplied by JNE. Third, JNE allows to the applications to inform to the execution environment about their processor requirements, as well as, JNE allows to the execution environment to answer to the applications with the number of processors assigned to them at any moment. Finally, JNE reduces the number of migrations of the processes allocated to an application, by using scheduling polices that consider multithreaded Java applications as an allocation unit.

5.2.3.1 Adaptive Java applications

The first issue considered in JNE is the capability of Java applications to adapt their behavior to the amount of resources offered by the execution environment (malleability [53]). The process is dynamic and implies three important aspects:

- ✓ First, the application should be able to request and release processors at any time. This requires from the execution environment an interface to set the number of processors the application wants to run.
- ✓ Second, the amount of parallelism that the application will generate (at a given moment) is limited by both the number of processors assigned to the application and by the amount of work pending to be executed. The execution

environment has to provide an interface to allow the application to check the number of processors available just before spawning parallelism.

- ✓ And third, the application should be able to react to processor preemptions and allocations resulting from the operating system allocation decisions. This requires mechanisms that allow the application, once informed, to recover from possible processor preemptions.

5.2.3.2 Application/JNE interface

Each Java application executing on the JNE shares information with the execution environment. The information includes the number of processors on which the application wants to run at any moment and the number of processors currently allocated by the execution environment to the application.

The interface between the applications and the JNE is implemented with a Java class called `jne`, which contains the following two Java methods for calling, through the Java Native Interface (JNI), the JNE services for requesting and consulting processors:

- ➢ `cpus_current()`: consult the current number of processors allocated to the invoking application.
- ➢ `cpus_request(num)`: request to the execution environment `num` processors.

5.2.3.3 JNE scheduler

The JNE scheduler is based on the Nanos RM mentioned in Section 6.4. It is responsible for the distribution of processors to applications. At any time, there is a current active scheduling policy that is applied to all applications running in the system. The scheduler observes application demands, estimates the load of the machine, and finally distributes processors accordingly. The scheduler also decides which processors are assigned to each application taking into account data affinity issues (i.e. helping the application to exploit data locality whenever possible).

JNE offers a set of scheduling policies, including batch, round robin, equipartition and others than combine space- and time-sharing. The evaluation in this thesis uses Dynamic Space Sharing (DSS) [119][120]. In DSS, each application receives a number of processors that is proportional to its request and inversely proportional to the total workload of the system, expressed as the sum of processor requests of all jobs in the system.

The JNE scheduler is implemented as a user-level process that wakes up periodically at a fixed time quantum, examines the current requests of the applications and distributes processors, applying a scheduling policy. With this configuration, direct modification of the native kernel is not required to show the usefulness of the proposed environment.

5.2.3.4 Self-adaptive JOMP applications

This thesis uses JOMP applications as the benchmark to evaluate the proposed mechanisms, as a particular case of multithreaded Java applications in HPC environments. In order to obtain self-adaptive JOMP applications, the implementation of the JOMP compiler and supporting runtime library has been modified to implement the communication between the application and JNE.

The JOMP runtime library has been modified so that, when an application starts, it requests as many processors for this application as indicated in one of the arguments of the interpreter command line (`-Djomp.threads`). This request is made using the `cpus_request()` method available in the JNE interface.

After that, every time the application has to spawn parallelism (i.e. at the beginning of each parallel region) the compiler injects a call to `cpus_current()` method from the JNE interface to check the number of processors currently allocated to the application. With this information, the application generates work for as many threads as processors available to run. This process can be appreciated in Figure 5.3.b, which shows the code generated by the JOMP compiler for the simple example shown in Figure 5.3.a highlighting the utilization of the JNE interface services.

The user can change the concurrency level of the application (to be used in the next parallel region) inside any sequential region invoking the `setNumThreads()` method from the JOMP runtime library. In this case, in order to inform the execution environment about the new processor requirements of the application, the JOMP compiler will replace this invocation with one to the `cpus_request()` method from the JNE interface.

```
public class Hello {
  public static void main (String argv[]) {
    int myid;
    //omp parallel private (myid)
    {
      myid = OMP.getThreadNum();
      System.out.println("Hello from" + myid);
    }
  }
}
```

(a) original code

```
public class Hello {
  public static void main (String argv[]) {
    int myid;
    // OMP PARALLEL BLOCK BEGINS
    jomp.runtime.OMP.setNumThreads(jne.cpus_current());
    {
      __omp_Class0 __omp_Object0 = new __omp_Class0();
      __omp_Object0.argv = argv;
      try {
        jomp.runtime.OMP.doParallel(__omp_Object0);
      } catch(Throwable __omp_exception) {
        jomp.runtime.OMP.errorMessage();
      }
      argv = __omp_Object0.argv;
    }
    // OMP PARALLEL BLOCK ENDS
  }
}

// OMP PARALLEL REGION INNER CLASS DEFINITION BEGINS
private static class __omp_Class0 extends jomp.runtime.BusyTask {
  String [] argv;
  public void go(int __omp_me) throws Throwable {
    int myid;
    // OMP USER CODE BEGINS
    {
      myid = OMP.getThreadNum();
      System.out.println("Hello from" + myid);
    }
    // OMP USER CODE ENDS
  }
}
// OMP PARALLEL REGION INNER CLASS DEFINITION ENDS
```

(b) transformed code

Figure 5.3. Example showing the use of the JNE interface for JOMP applications

5.2.3.5 Nano-threads library (NthLib)

The Nano-threads Library [101] (NthLib) is a user level threads package specially designed for supporting parallel applications. The role of NthLib is two fold. On one hand, NthLib provides the user level execution environment in which applications execute. On the other hand, NthLib cooperates with the execution environment by interchanging significant fine grain information on accurate machine state and resource utilization, throughout the execution of the parallel application.

NthLib provides the following services:

- Thread management services: `nth_create` (create nano-thread), `nth_exit` (finalize nano-thread), `nth_wait` (block nano-thread) and `nth_yield` (yield virtual processor to another nano-thread).
- Generic queue management services: `nth_queue_init` (initialize queue), `nth_enqueue`/`nth_dequeue` (enqueue/dequeue nano-thread on/from queue).
- Ready queue management services: `nth_to_rq` (enqueue nano-thread on global ready queue) and `nth_to_lrq` (enqueue nano-thread on local ready queue).
- Mutual exclusion services: `spin_init` (initialize spin), `spin_lock` (lock spin) and `spin_unlock` (unlock spin).

5.2.3.6 Implementation of JNE

As commented before, the JVM implementation of SGI Irix implements the native threads model using the Pthreads library (Figure 5.4.a). In order to implement the mechanisms described in Section 5.2.3, the Pthreads library has been replaced with the NthLib library. This replacement technique makes JNE portable to all platforms where NthLib is available. In order to avoid modifications of the JVM, the Pthreads library interface is maintained but the library methods have been rewritten using the services provided by NthLib (Figure 5.4.b).

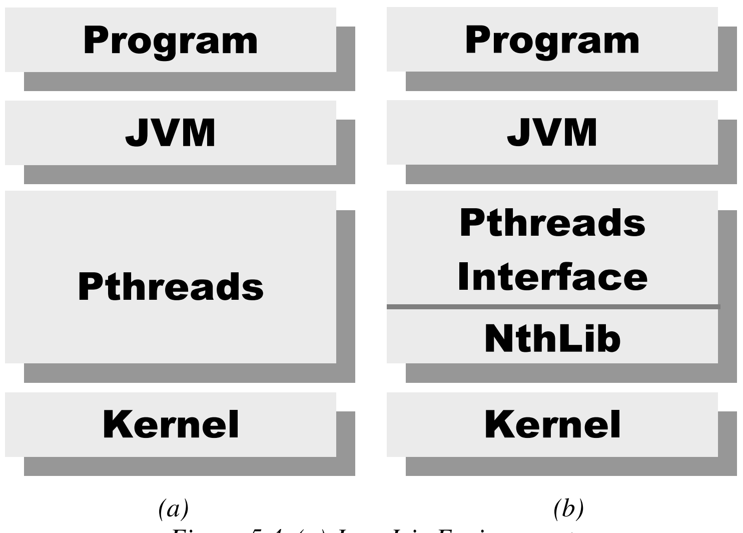


Figure 5.4. (a) Java Irix Environment
(b) Java Nanos Environment

5.2.3.6.1 NthLib implementation basics

Each virtual processor (i.e. kernel thread) has an idle thread that runs when there is not useful work for execute. This idle thread is responsible of looking for new work to execute, by accessing to the local ready queue of this virtual processor, and if no work is found, accessing to global ready queue. This idle thread also executes periodically a function for dequeuing from the queues of alarms all the elapsed alarms (see `pthread_cond_timedwait` function implementation). Finally, the idle threads also collaborate with the JNE scheduler for managing the processor preemptions.

NthLib services are implemented using the functions provided by the Quick Threads package [92].

5.2.3.6.2 Pthread creation and destruction

- ✓ `int pthread_create (pthread_t *thread, const pthread_attr_t *attr,`
  `void *(*start)(void *), void *arg)`

This function creates a new pthread. As one pthread maps on one nano-thread, this function creates one nano-thread using the `nth_create` service of NthLib. The function initializes all the information from this pthread (state, identifier, signal queue, signal mask, pthread keys, attributes, etc). All this pthread private data is stored in the user data area of the nano-thread associated with this pthread. Finally, this function adds the nano-thread to the global ready queue using the `nth_to_rq` service of NthLib.

- ✓ `void pthread_exit (void *retval)`

This function destroys the invoking pthread using the `nth_exit` service of NthLib.

5.2.3.6.3 Pthread mutex implementation

Each mutex has associated a counter, a spin lock and a queue where nano-threads block waiting for accessing this mutex. The spin lock is operated using the mutual exclusion services provided by NthLib and the queue is operated with the generic queue management services provided by NthLib.

- ✓ `int pthread_mutex_lock (pthread_mutex_t *mutex)`

The nano-thread executing this function acquires the spin lock and checks the counter associated to the mutex. If the counter is greater than zero, the nano-thread unlocks the spin, adds itself to the queue of this mutex and then blocks executing the `nth_wait` service of NthLib. The nano-thread remains blocked in this function until a `pthread_mutex_unlock` is performed on to this mutex. When this occurs, the nano-thread continues its execution by returning from the `nth_wait` function, and repeats the previous process until the counter associated to the mutex is zero. In this case, the nano-thread increments by one the counter, unlocks the spin and returns.

- ✓ `int pthread_mutex_trylock (pthread_mutex_t *mutex)`

The nano-thread executing this function acquires the spin lock and checks the counter associated to the mutex. If the counter is greater than zero, the nano-thread unlocks the spin, and returns indicating that the mutex is busy. If the counter associated to the mutex is zero, the nano-thread increments by one the counter, unlocks the spin and returns.

- ✓ `int pthread_mutex_unlock (pthread_mutex_t *mutex)`

The nano-thread executing this function acquires the spin lock, decrements counter associated to the mutex by one and then checks if it is zero. If this occurs, it dequeues the first nano-thread waiting in the queue of this mutex and adds it to the global ready queue using the `nth_to_rq` service of NthLib.

5.2.3.6.4 Pthread conditional variables implementation

Each conditional variable has a queue where nano-threads block waiting for a notification in the conditional variable. This queue is operated with the generic queue management services provided by NthLib.

- ✓ `int pthread_cond_signal (pthread_cond_t *cond)`

This function dequeues the first nano-thread waiting in the queue of this conditional variable and adds it to the global ready queue using the `nth_to_rq` service of NthLib.

- ✓ `int pthread_cond_broadcast (pthread_cond_t *cond)`

This function dequeues all the nano-threads waiting in the queue of this conditional variable and adds them to the global ready queue using the `nth_to_rq` service of NthLib.

✓ `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)`

The nano-thread executing this function adds itself to the queue of this conditional variable, releases the mutex associated to this conditional variable and then blocks executing the `nth_wait` service of NthLib. The nano-thread remains blocked in this function until a notification is sent to this conditional variable. When this occurs, the nano-thread continues its execution by returning from the `nth_wait` function, reacquires the mutex associated to this conditional variable and returns.

✓ `int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)`

This function requires the implementation of temporization. Each virtual processor has associated a timer and a queue of pending alarms to be generated using this timer. This queue is ordered depending on the absolute time in which each alarm must be generated. The services to operate on this queue have been implemented following the same semantics that the generic queue management services provided by NthLib. At any time, the timer is counting the time left to generate the first alarm of the queue of alarms. When the timer is elapsed, the timer handler reprograms the timer with the next alarm in the queue of alarms, and indicates (setting a global variable) that there are elapsed alarms in the queue of alarms.

When a nano-thread executes the `pthread_cond_timedwait` function, it adds a new pending alarm to the queue of alarms of the current virtual processor and reprograms the timer is the added alarm must be generated before the alarm that is currently programmed in the timer. Then the nano-thread adds itself to the queue of this conditional variable, releases the mutex associated to this conditional variable and then blocks executing the `nth_wait` service of NthLib. The nano-thread remains blocked in this function until a notification is sent to this conditional variable or the programmed alarm is generated. When this occurs, the nano-thread continues its execution by returning from the `nth_wait` function, reacquires the mutex associated to this conditional variable and returns.

The idle thread on each virtual processor executes periodically a function that, when the global variable indicates that there are elapsed alarms at any queue of alarms, is responsible of eliminate these alarms from the queue, dequeue the nano-threads that programmed these alarms from the queue of the conditional variable and add them to the global ready queue using the `nth_to_rq` service of NthLib.

5.2.3.6.5 Pthread signals implementation

Each pthread has a queue of pending signals. The services to operate on this queue have been implemented following the same semantics that the generic queue management services provided by NthLib.

Every time that a pthread enters the system (by invoking a pthreads library service) or returns to user level after being blocked within the system, it checks its signals queue looking for pending signals to be processed. Each signal in this queue is sent to the pthread using the `kill` system call.

- ✓ `int pthread_kill (pthread_t thread, int sig)`

The nano-thread executing this function finds the nano-thread with the pthread identifier passed as parameter. If this nano-thread is running and the signal is not masked, the signal is sent using the `kill` system call. Otherwise, a new signal is queued in the pending signals queue of the pthread associated to this nano-thread. If the nano-thread was blocked in a conditional variable or in a mutex, it is removed from the queue and added to the global ready queue using the `nth_to_rq` service of NthLib.

5.2.4 Evaluation

This section presents the main conclusions drawn from the experimentation with JNE using the Java Grande Benchmarks described in Section 2.2.4.1.

Although JNE has been developed to improve performance of malleable applications (that is, applications able to adapt their behavior to the amount of resources offered by the execution environment), it is desirable that JNE supports the efficient execution of non-malleable applications too, which are common (and often it is not easy convert them to malleable). For example, in the JOMP version of the Java Grande Benchmarks, only SOR, LUFact and Euler are malleable. Crypt, Series, MonteCarlo and RayTracer are not malleable because they only have one parallel region and, as commented in Section 5.2.3.4, adaptability is achieved at the beginning of each parallel region. Sparse is not malleable because the concurrency level of the application is used as size in some data structures, making impossible to change dynamically this value without modifying the application.

This evaluation includes experiments with malleable applications based on SOR and LUFact and experiments with non-malleable applications based on Crypt and Series.

5.2.4.1 Single application performance

In the first set of experiments, an individual instance of SOR, LUFact, Crypt and Series is executed inside a *cpuset*, in its sequential version and its JOMP version with different concurrency levels (between 1 and 16 threads). This experiment intends to evaluate the impact on performance of the Pthreads library replacement by the NthLib, and analyze the scalability of each application.

The speedup obtained for SOR, LUFact, Crypt and Series running with different concurrency levels with respect to the sequential version is plotted in Figure 5.5, Figure 5.6, Figure 5.7 and Figure 5.8, respectively. For each experiment, 10 executions have been performed. The first series (labeled *IRIX*) corresponds to the execution on the native Irix system. The second series (labeled *IRIX+SETC*) corresponds to the execution on the native Irix system when the application informs to the Pthreads library about its concurrency level (using the mechanism described in Section 5.2.2). The third series (labeled *JNE–CPUM*) corresponds to the execution time on top of the JNE with the JNE scheduler not active. And the fourth series (labeled *JNE+CPUM*) corresponds to the execution time on top of the JNE with the JNE scheduler active.

From the analysis of the speedup figures of malleable applications (SOR and LUFact, Figure 5.5 and Figure 5.6, respectively) four important conclusions can be derived. First, the performance obtained running with *IRIX* is very low, due to the large number of process migrations occurred. For example, for LUFact with concurrency level of 8 threads the system performs 9.6 process migrations per second on average. An important part of these process migrations are produced when application invokes the `yield()` method. The Pthreads library does not try to exploit any data affinity in this point, and relies on the underlying operating system to perform the yield operation. This increases the process migrations and reduces data affinity. This problem acquires special relevance in JOMP applications (especially when they have several parallel regions), which frequently use the `yield()` method (when threads look for new work to be executed or when threads wait for a barrier to be opened), like many others runtimes do, to implement a polite scheduling that allows others threads to execute when there is not useful work to do.

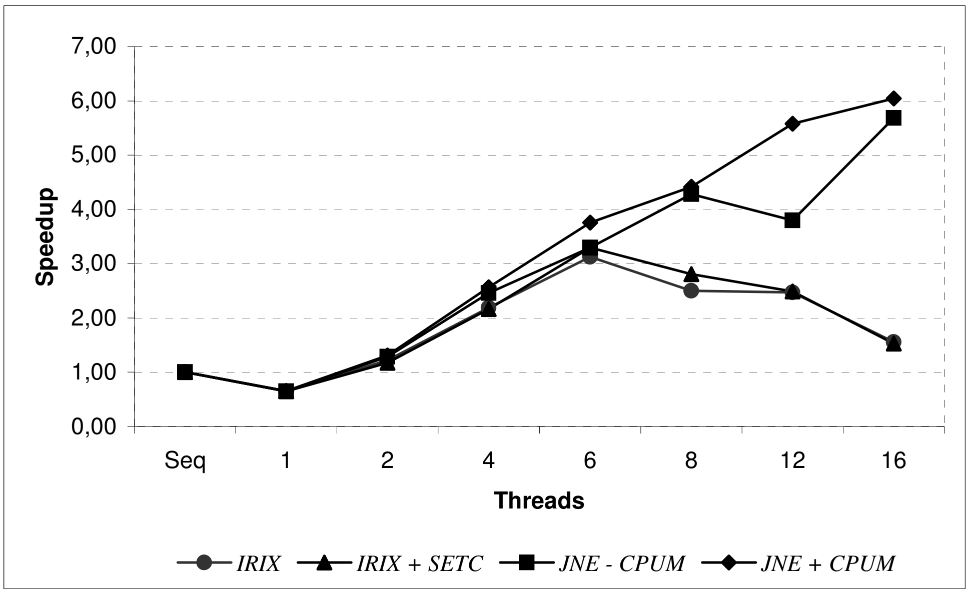


Figure 5.5. SOR standalone speedup

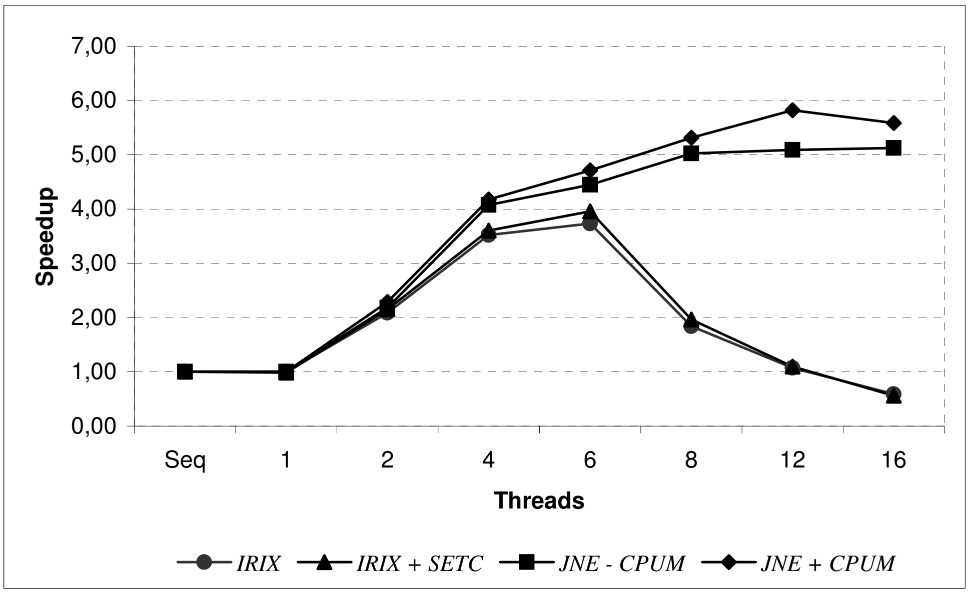


Figure 5.6. LUFact standalone speedup

The second conclusion is that, as it has been advanced in Section 5.2.2, improvements on performance when running with *IRIX+SETC* are not very high, because the JOMP runtime creates threads at user level with a long lifetime. However, the large number of migrations performed by Irix is still the main cause of the bad behavior.

The third conclusion is that running with *JNE–CPUM* provides noticeable performance improvements that can be explained as follows. NthLib tries to exploit data affinity itself at nano-thread level. When a thread invokes the `nth_yield()` method, it yields its kernel thread to another nano-thread and enqueues itself in the local ready queue of this kernel thread. In this way, data affinity at nano-thread level is improved and the yield operation is accomplished avoiding unnecessary operating system intervention, reducing the number of process migrations (1.4 process migrations per second on average when executing LUFact with concurrency level of 8 threads).

Notice that *JNE–CPUM* does not bind kernel threads to processors in the *cpuset*. This explains the anomalous behavior observed for 6 and 12 threads. In both cases, the application is executed in a *cpuset* larger than the number of processors required (*cpuset* of 8 processors and *cpuset* of 16 processors, respectively). This means that there are free processors, and as kernel threads are not bound with processors, migrations are incremented (11.6 migrations per second on average when executing SOR with concurrency level of 12 threads).

The last conclusion of this set of experiments is that running with *JNE+CPUM* improves the performance even more. In addition to all the advantages of the *JNE–CPUM* approach, the JNE scheduler strengthens data affinity at kernel thread level by binding kernel threads to the processors assigned to the application. This binding totally eliminates process migrations.

The low scalability achieved in these applications can be explained because SOR and LUFact have one parallel region repeated several times inside a time step loop. This means that work generation and thread synchronization are done several times, both facts producing considerable overhead. Besides, threads reuse data at every parallel region, so process migrations can heavily affect performance because data affinity is lost.

On the other side, the analysis of speedup figures of non-malleable applications (Crypt and Series, Figure 5.7 and Figure 5.8, respectively) reveals that all the approaches evaluated obtain similar performance, achieving good scalability (nearly linear). Only when running with *IRIX* the speedups obtained are a little bit worse because the execution environment (Pthreads library in this case) needs some time to estimate the concurrency level of the application, how it has been explained in

Section 5.2.2. Notice that, if the execution environment is informed about this concurrency level, as it is done in the other approaches, performance is improved.

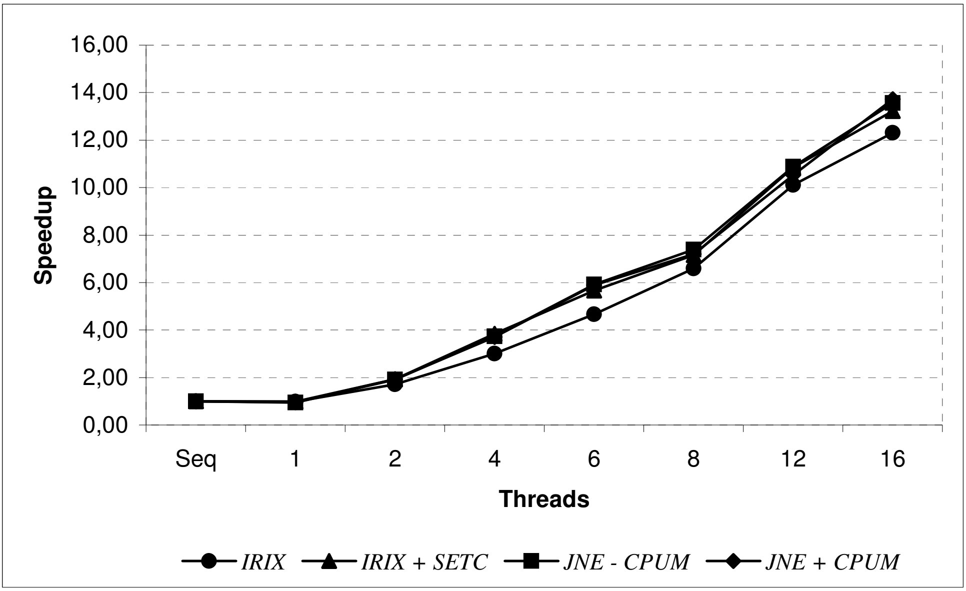


Figure 5.7. Crypt standalone speedup

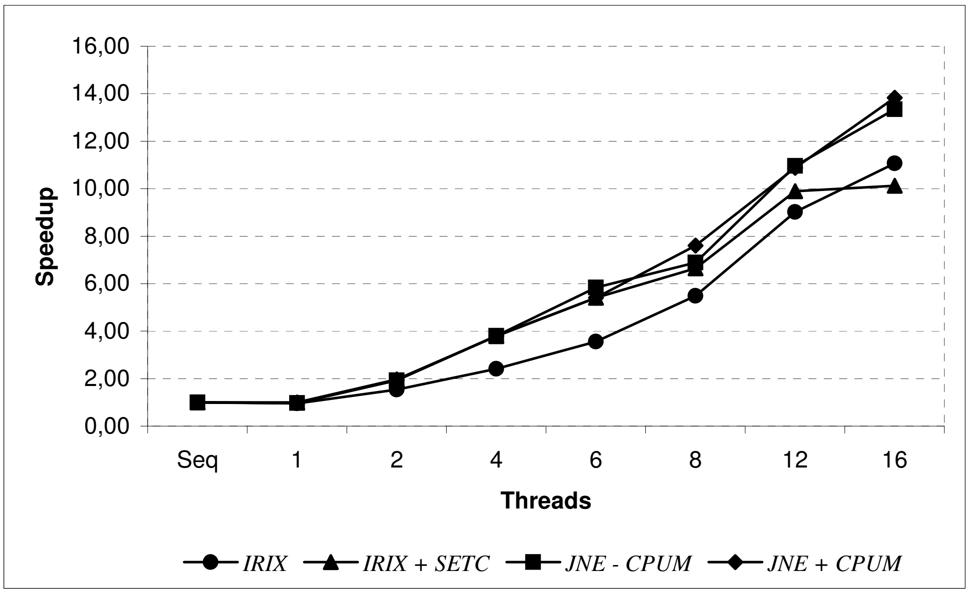


Figure 5.8. Series standalone speedup

The high scalability achieved in these applications can be explained because Crypt and Series have only one parallel region. This means that work generation and thread synchronization are done only once, minimizing the overhead produced. Besides, threads do not reuse data, so process migrations are not critical for performance.

5.2.4.2 Multiprogrammed workloads

5.2.4.2.1 Malleable applications

For the second set of experiments, a workload composed of an instance of LUFact with concurrency level of 2 threads, an instance of SOR with concurrency level of 4 threads, an instance of LUFact with concurrency level of 4 threads and an instance of SOR with concurrency level of 6 threads has been defined. These applications instances are simultaneously started inside a *cpuset* with 16 processors, and they are continuously restarted until one of them is repeated 10 times. Notice that the system is not overloaded (i.e. the number of processors in the *cpuset* is greater or equal than the maximum load). This experiment intends to evaluate the performance of JOMP malleable applications in a non-overloaded multiprogrammed environment.

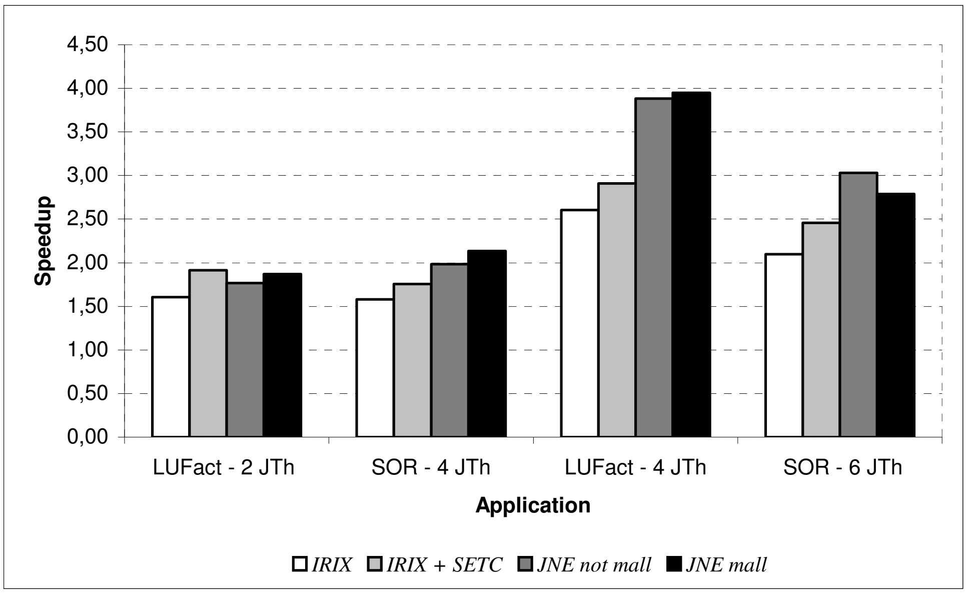


Figure 5.9. Application speedups in the 1st workload (non-overloaded environment – malleable applications)

Figure 5.9 draws the speedup obtained for each application instance in the workload relative to the sequential version. The first and second series have the same meaning as in the first set of experiments. The third series (labeled *JNE not mall*) corresponds to the execution time on top of the JNE with the JNE scheduler active using a DSS scheduling policy, assuming that applications do not use the JNE interface to adapt themselves to the available resources (they behave as non-malleable applications). And the fourth series (labeled *JNE mall*) corresponds to the execution time on top of the JNE when applications use the JNE interface to adapt themselves to the available resources.

Since the system is not overloaded, each application instance should be able to run with as many processors as requested. Therefore, the speedup should be the same as if executed alone in the *cpuset*. However, speedup figures are worse than one could expect.

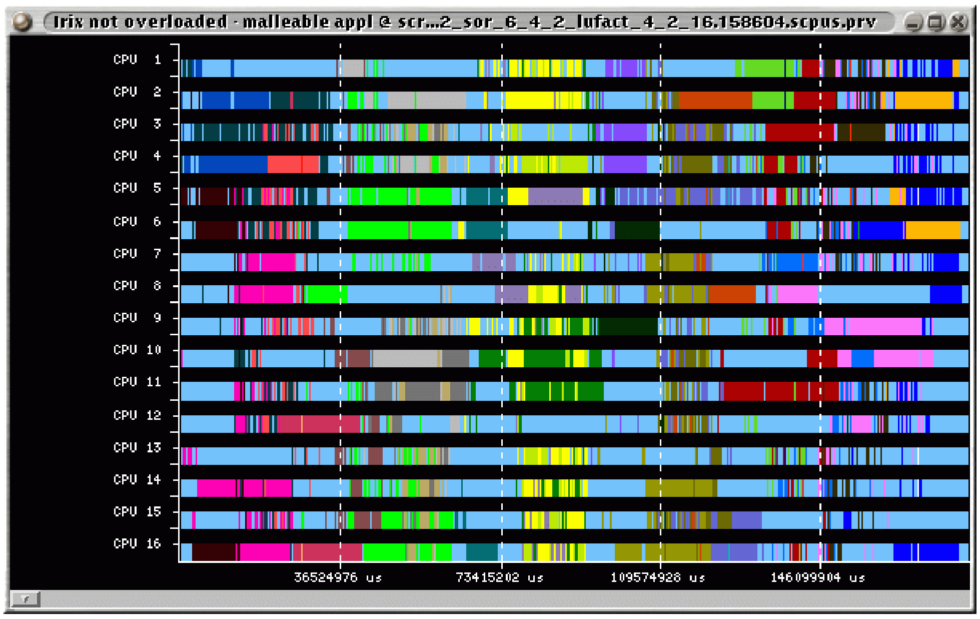


Figure 5.10. Process migrations when running with Irix in the 1st workload (non-overloaded environment – malleable applications)

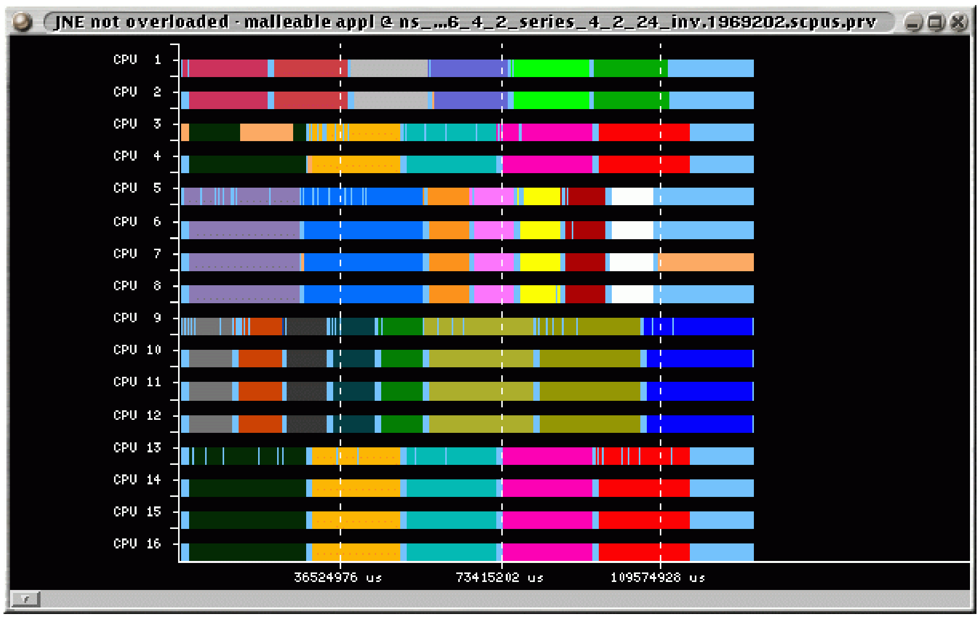


Figure 5.11. Process migrations when running with JNE in the 1st workload (non-overloaded environment – malleable applications)

First, when executing with *IRIX*, the speedup achieved is very low. This is caused by the continuous process migrations that reduce considerably the data reuse. In this experiment, these process migrations have been quantified in 13 migrations per second on average. These process migrations can be appreciated in Figure 5.10, which shows a Paraver window in which each color represents the execution of an application instance. Second, running with *IRIX+SETC* improves the speedup achieved (because of the effect commented in Section 5.2.2). However, the same scheduling problems of *IRIX* are not solved.

Third, notice that important improvements are obtained when running with JNE. This is caused by two factors: the inherent benefits of using JNE demonstrated in Section 5.2.4.1, and the action of the JNE scheduler in a multiprogrammed workload. In this case, the JNE scheduler binds kernel threads to processors, avoiding unnecessary process migrations and allowing more data reuse. In addition to this, the use of equitable policies like DSS makes possible that all applications instances in the workload get resources, avoiding application starvation or very unbalanced executions. This behavior can be appreciated in Figure 5.11.

Considering the observed behavior, the only question is why application instances running with JNE in the workload do not achieve the speedup of their counterparts running alone. The answer to this question is the interference produced when running in *cpusets* as mentioned in Section 2.2.4.2.

Finally, notice that in a non-overloaded system it is not important if applications are malleable, because there are enough resources to satisfy all the requests. Therefore, it is not necessary that applications adapt themselves.

Table 5.2. Performance degradation of each application instance in the 1st workload vs. best standalone execution

Application	IRIX	IRIX + SETC	JNE not mall	JNE mall
LUFact 2 JTh	0.70	0.85	0.77	0.82
SOR 4 JTh	0.59	0.67	0.77	0.83
LUFact 4 JTh	0.62	0.71	0.93	0.95
SOR 6 JTh	0.53	0.64	0.81	0.74

These conclusions are consolidated in Table 5.2, which summarizes the observed performance degradation for each application instance in this workload with respect to best standalone execution. Performance degradation is calculated dividing the best application standalone execution time by the average execution time of an application instance in a workload.

For the third set of experiments, a workload composed of an instance of LUFact with concurrency level of 4 threads, an instance of SOR with concurrency level of 8 threads, an instance of LUFact with concurrency level of 8 threads and an instance of SOR with concurrency level of 12 threads has been defined. These applications instances are simultaneously started on a *cpuset* with 16 processors, and they are continuously restarted until one of them is repeated 10 times. Notice that, the maximum load is 32, which is higher than the number of processors available in the *cpuset*, so the system is overloaded. This experiment intends to evaluate the performance of JOMP malleable applications when running in an overloaded multiprogrammed environment.

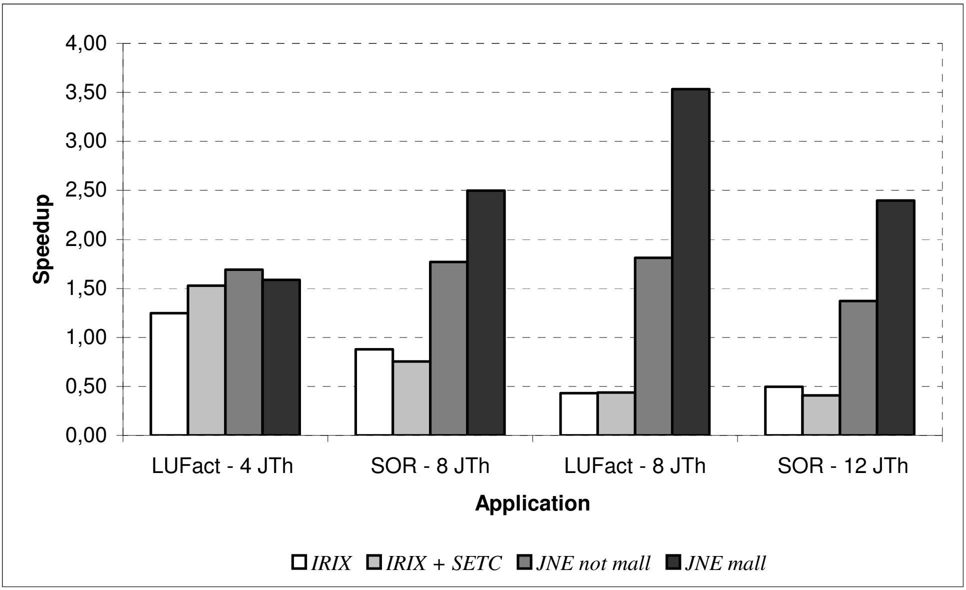


Figure 5.12. Application speedups in the 2nd workload (overloaded environment – malleable applications)

Figure 5.12 draws the speedup for each application instance in the workload relative to the sequential version. All the series have the same meaning as in the previous workload. Since the system is overloaded, each application instance receives fewer processors than requested (as many processors as assigned by the DSS policy in the JNE scheduler). Therefore, the speedup should be the same as if executed alone in the *cpuset* with the number of processors allocated by the JNE scheduler.

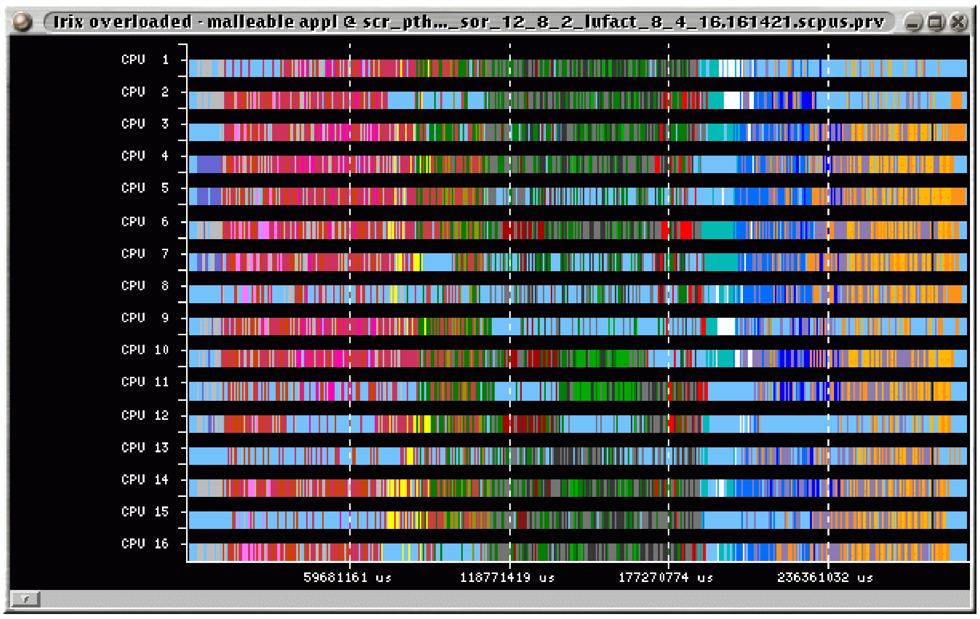


Figure 5.13. Process migrations when running with Irix in the 2nd workload (overloaded environment – malleable applications)

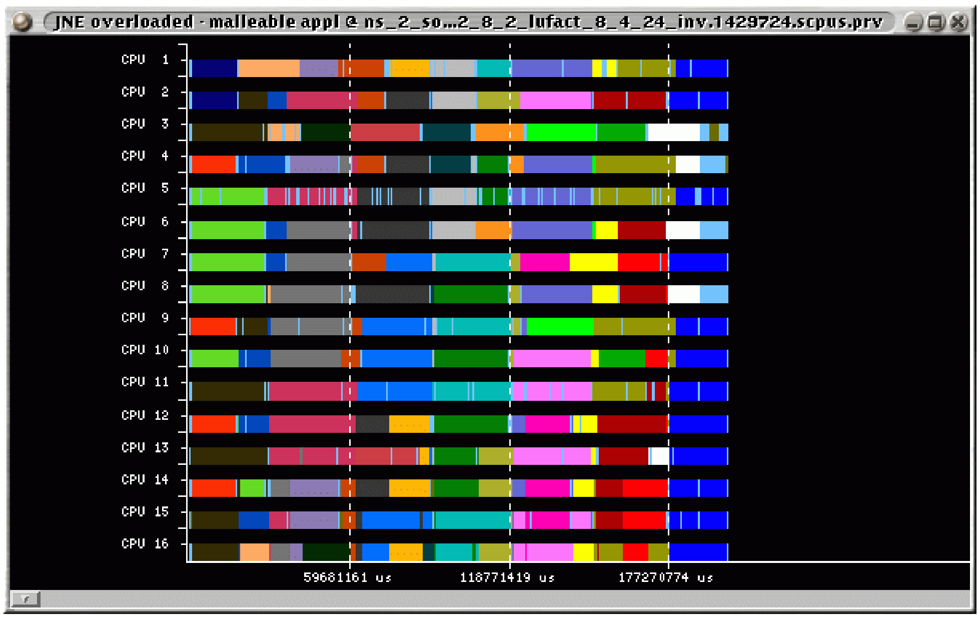


Figure 5.14. Process migrations when running with JNE in the 2nd workload (overloaded environment – malleable applications)

All the conclusions exposed for the first workload are valid also in this case. In addition, some considerations must to be taken into account. First, the continuous process migrations when executing with *IRIX* have been incremented even more (43.9 process migrations per second on average), as shown in Figure 5.13. In addition to this, notice that the Irix scheduling causes a noticeable unbalanced execution (benefits some applications and damages others). For example, in this case LUFact with concurrency level of 4 threads is receiving proportionally more resources than the other application instances. For this reason, its performance degradation is lower.

When running with *JNE*, the action of the JNE scheduler in an overloaded multiprogrammed workload is even more important. A rational use of the resources allows the reduction of processor migrations (0.9 process migrations per second on average) allowing better locality exploitation and a balanced execution of all the application instances of the workload, as shown in Figure 5.14. Like in the first workload, the interference produced when running in *cpusets* causes application instances not to achieve the expected speedup. Besides, other factors as processor preemptions overhead or the processor distribution algorithm itself, can influence the speedup obtained.

Notice that, in an overloaded system it is very important if applications are malleable, because there are not enough resources to satisfy all the requests. Malleability reduces the number of Java threads created by the application thus reducing the overheads incurred in the parallel execution and management of threads. Figure 5.12 shows that the speedup achieved with *JNE mall* approaches the speedup of using half the number of threads (as assigned by the DSS policy in this scenario).

Table 5.3. Performance degradation of each application instance in the 2nd workload vs. best standalone execution

Application	IRIX	IRIX + SETC	JNE not mall	JNE mall
LUFact 4 JTh	0.30	0.37	0.40	0.38
SOR 8 JTh	0.19	0.17	0.40	0.57
LUFact 8 JTh	0.08	0.08	0.34	0.66
SOR 12 JTh	0.08	0.07	0.25	0.43

Table 5.3 summarizes the observed performance degradation for each application instance in the second workload with respect to best standalone execution. Notice that the results confirm the benefit obtained when running multiprogrammed workloads with JNE, and the convenience of using malleable applications able to adapt themselves to the available resources.

5.2.4.2.2 Non-malleable applications

For the fourth set of experiments, a workload composed of an instance of Series with concurrency level of 2 threads, an instance of Crypt with concurrency level of 4 threads, an instance of Series with concurrency level of 4 threads and an instance of Crypt with concurrency level of 6 threads has been defined. These applications instances are simultaneously started inside a *cpuset* with 16 processors, and they are continuously restarted until one of them is repeated 10 times. This experiment intends to evaluate the performance of JOMP non-malleable applications in a non-overloaded multiprogrammed environment.

Notice that with non-malleable applications, the adaptation to the available resources is done only once, at the beginning of the only parallel region, and maintained during the entire region. This fact can lead to have unused processors if the application receives more processors while it is executing inside the parallel region, because at this point the application cannot generate new parallelism to run at these processors. In order to avoid this situation, non-malleable applications use the JNE interface to adapt their concurrency level to the double of the available resources (*JNE mall* in Figure 5.15 and Figure 5.18).

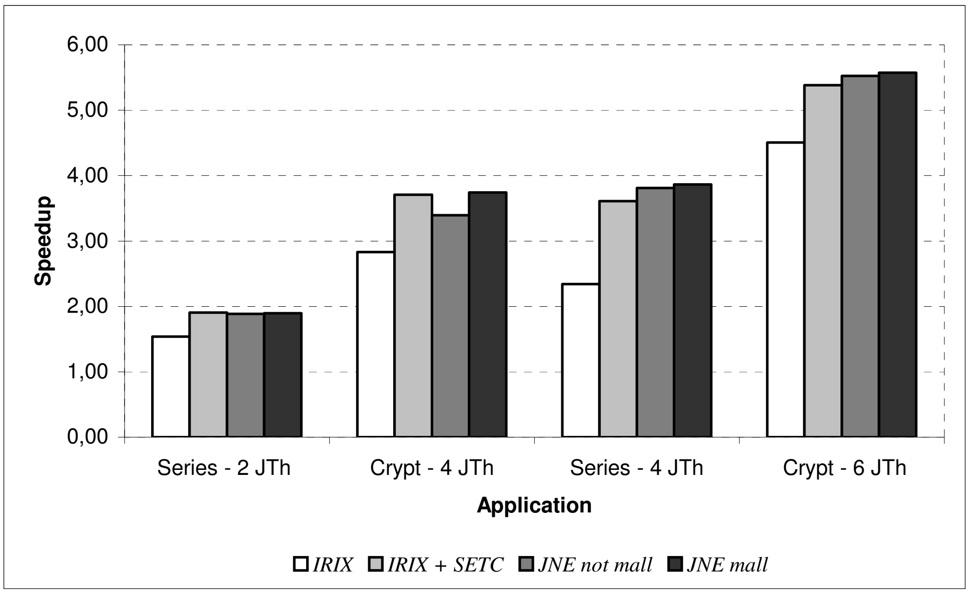


Figure 5.15. Application speedups in the 3rd workload (non-overloaded environment – non-malleable applications)

Figure 5.15 draws the speedup for each application instance in the workload relative to the sequential version. Instead of *JNE mall*, all the series have the same meaning as in the previous workload. Since the system is not overloaded, each application instance should be able to run with as many processors as requested.

Therefore, the speedup should be the same as if executed alone in the *cpuset*. The results obtained verify this theory.

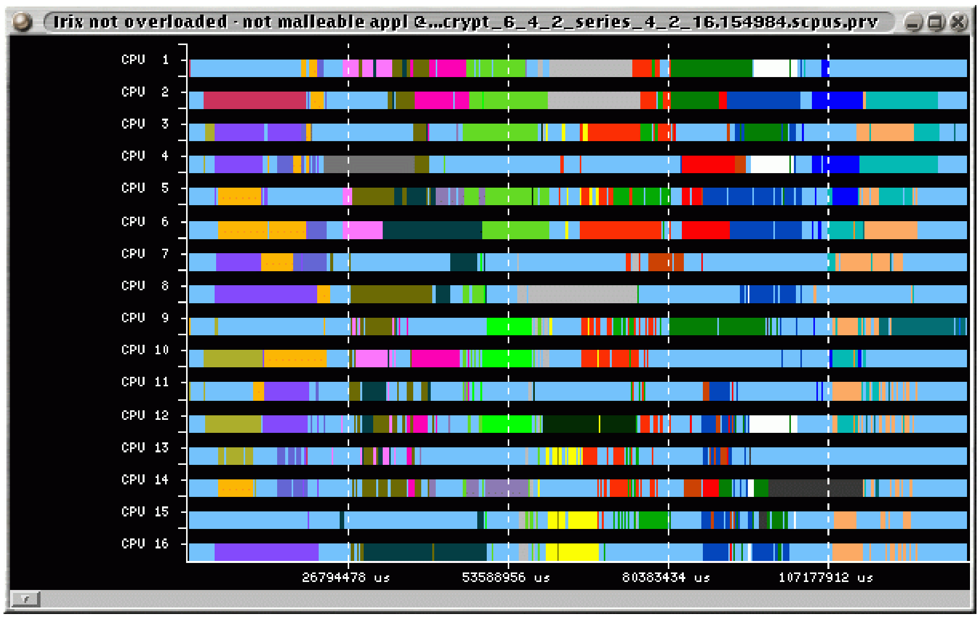


Figure 5.16. Process migrations when running with Irix in the 3rd workload (non-overloaded environment – non-malleable applications)

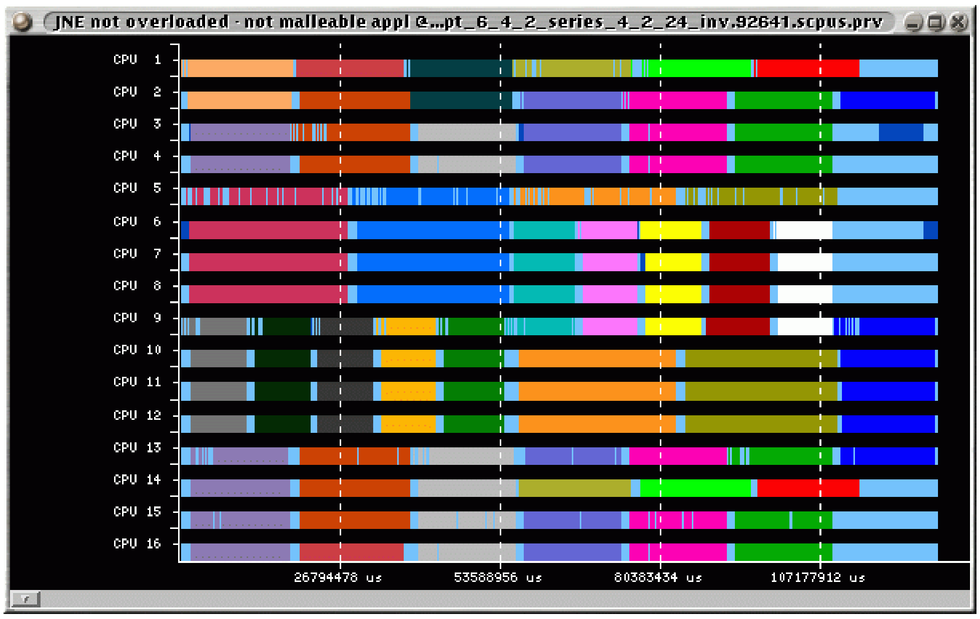


Figure 5.17. Process migrations when running with JNE in the 3rd workload (non-overloaded environment – non-malleable applications)

Notice that, as commented in Section 5.2.4.1, in this kind of applications process migrations (which can be appreciated in Figure 5.16 when running with Irix and in Figure 5.17 when running with JNE) are not critical for performance (when running with *IRIX+SETC* 6.4 migrations per second on average have been measured without detecting any performance degradation).

Table 5.4. Performance degradation of each application instance in the 3rd workload vs. best standalone execution

Application	IRIX	IRIX + SETC	JNE not mall	JNE mall
Series 2 JTh	0.78	0.97	0.93	0.93
Crypt 4 JTh	0.74	0.97	0.90	0.99
Series 4 JTh	0.62	0.95	0.98	0.99
Crypt 6 JTh	0.79	0.94	0.98	0.99

This experiment confirms that in a non-overloaded system it is not important if applications adapt their behavior to the available resources, because there are enough resources to satisfy all the requests. These conclusions are consolidated in Table 5.4, which summarizes the observed performance degradation for each application instance in the third workload with respect to best standalone execution.

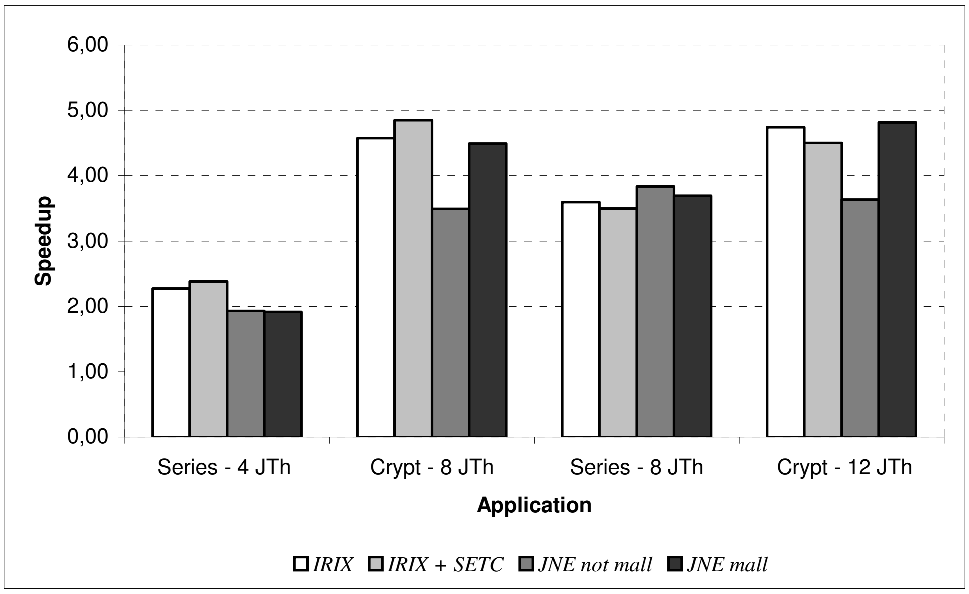


Figure 5.18. Application speedups in the 4th workload (overloaded environment – non-malleable applications)

For the last set of experiments, a workload composed of an instance of Series with concurrency level of 4 threads, an instance of Crypt with concurrency level of 8 threads, an instance of Series with concurrency level of 8 threads and an instance of Crypt with concurrency level of 12 threads has been defined. These applications instances are simultaneously started on a *cpuset* with 16 processors (the system is overloaded), and they are continuously restarted until one of them is repeated 10 times. This experiment intends to evaluate the performance of JOMP non-malleable applications when running in an overloaded multiprogrammed environment.

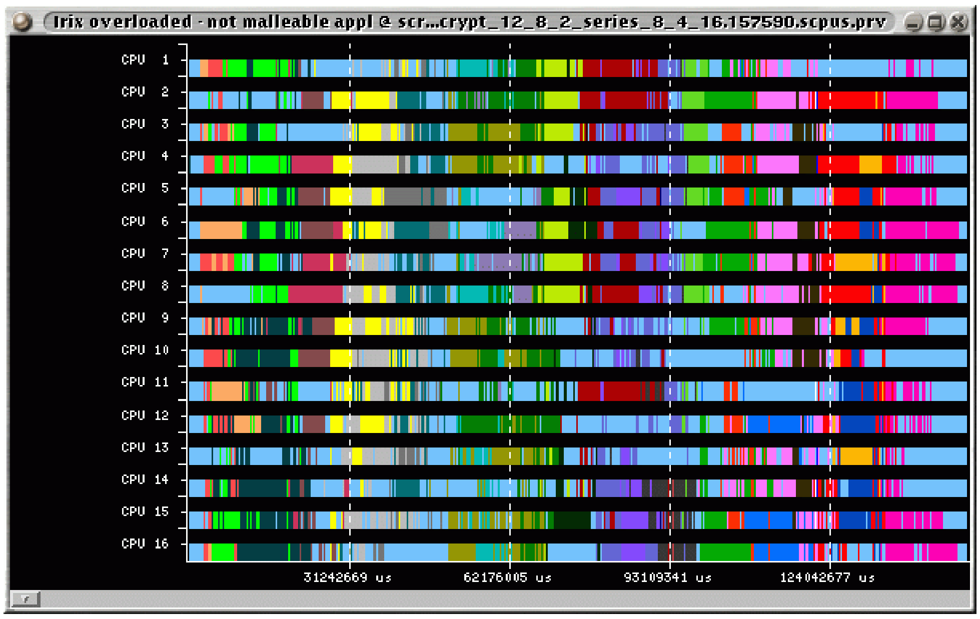


Figure 5.19. Process migrations when running with Irix in the 4th workload (overloaded environment – non-malleable applications)

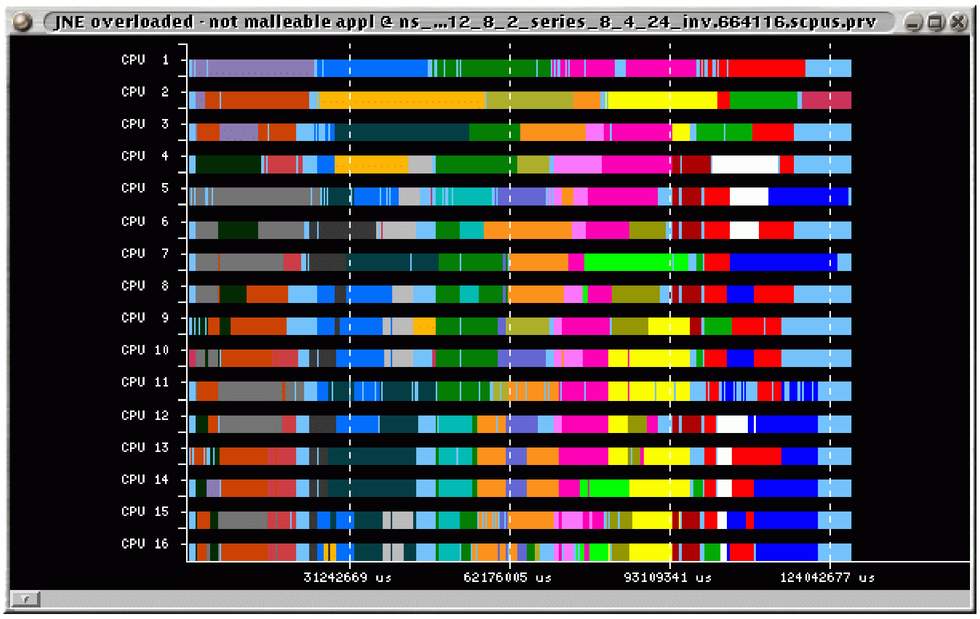


Figure 5.20. Process migrations when running with JNE in the 4th workload (overloaded environment – non-malleable applications)

Figure 5.18 draws the speedup for each application instance in the workload relative to the sequential version. All the series have the same meaning as in the previous workload. Since the system is overloaded, each application instance will receive fewer processors than requested (as many processors as assigned by DSS policy in the JNE scheduler). Therefore, the speedup should be the same as if executed alone in the *cpuset* with the number of processors allocated by the JNE scheduler. The results obtained in this workload verify this theory and confirm that performance obtained when running with JNE and generating as parallelism as the double of the resources assigned (*JNE mall*) is comparable to the one obtained when running with the original system. The execution of the different application instances can be appreciated in Figure 5.19 when running with Irix and in Figure 5.20 when running with JNE.

Finally, notice the performance degradation produced when running with *JNE not mall*, because the applications do not adapt to the available resources (they may have unused processors if the number of threads created is not multiple of the number of processors assigned to them). Table 5.5 shows the performance degradation of each application instance in the fourth workload with respect to best standalone execution.

Table 5.5. Performance degradation of each application instance in the 4th workload vs. best standalone execution

Application	IRIX	IRIX + SETC	JNE not mall	JNE mall
Series 4 JTh	0.60	0.63	0.49	0.49
Crypt 8 JTh	0.61	0.65	0.47	0.61
Series 8 JTh	0.49	0.47	0.50	0.49
Crypt 12 JTh	0.43	0.41	0.33	0.44

5.3 Resource Provisioning for Multithreaded Java Application Servers in e-Business Environments

5.3.1 Dynamic Resource Provisioning

Due to the great variability of the workloads of web applications (including unexpected flash crowds), it is difficult to estimate application resource requirements in advance, and hence provisioning resources to web applications in a hosting platform is problematic. Static allocation of resources can result in significant performance degradation when loads exceed capacity if under-provisioning has been performed, while over-provisioning resources based on worst-case workload estimation can result in poor resource utilization.

Recent studies have shown the considerable benefits of dynamically reallocate resources among hosted applications based on the variations in their workloads instead of over-provisioning resources in a hosting platform [7][35][36]. The goal is to meet the application requirements on demand and adapt to their changing resource needs. In this way, better resource utilization by extracting multiplexing gains can be achieved and the system can react to unexpected workload increases.

This thesis proposes a dynamic resource provisioning mechanism based on a global processor manager, called eDragon CPU Manager (ECM), responsible of distributing periodically the available processors among applications in a hosting platform applying a given policy. The ECM cooperates with the applications to manage efficiently the processors using a bi-directional communication. On one side, the applications request periodically to the ECM the number of processors needed to handle their incoming load avoiding the QoS degradation. On the other side, the ECM can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications can apply an admission control mechanism to limit the number of admitted requests accepting only those that can be served with the allocated processors without degrading their QoS (see Section 4.3.2).

5.3.1.1 Applications/ECM communication

The communication between the ECM and the applications is implemented using a shared memory region. The shared information includes the number of processors on which the application wants to run at any moment and the number of processors currently allocated by the ECM to the application. In order to manipulate this information, an interface between the applications and the ECM has been defined. This interface contains the following two Java methods accessible through the Java Native Interface (JNI):

- `cpusAssigned()`: consult the current number of processors allocated to the invoking application.
- `cpusRequested(num)`: request to the execution environment `num` processors.

In order to be self-managed, applications must be able to determine the number of processors they need to handle their incoming workload avoiding QoS degradation. This thesis achieves this capability by adding an internal manager within the server that runs the web application. This manager continuously monitors the number of incoming connections by performing online measurements distinguishing new SSL connections from resumed SSL connections. Based on the number of incoming new SSL connections, the number of incoming resumed SSL connections and the estimated computing demand of each kind of connection, the manager periodically calculates the number of processors needed to handle these connections maintaining their QoS and informs to the ECM using the `cpusRequested` method.

5.3.1.2 eDragon CPU manager (ECM)

The eDragon CPU Manager (ECM) is responsible for the distribution of processors among applications in the hosting platform. The ECM processes all the applications requests and distributes processors according to a given policy. Traditionally, resource allocation policies have considered conventional performance metrics such as response time, throughput and availability. However, the metrics that are of utmost importance to the management of an e-commerce site are revenue and profits and should be incorporated when designing policies [102].

The ECM can be configured to implement different policies, depending on the hosting platform needs, considering conventional performance metrics as well as incorporating e-business indicators. As an example, this thesis includes the implementation of a policy that considers customers of different priority classes (such as Gold, Silver or Bronze). The priority class indicates a customer domain's priority in relation to other customer domains. It is expected that high priority customers will receive preferential service respect low priority customers. In the policy, each application receives a number of processors ($Assig_i$) that is proportional to its request (Req_i) pondered depending of the application priority class ($Prio_i$) and inversely proportional to the total workload of the system ($\Sigma\, Prio_j * Req_j$), expressed as the sum of requests of all applications in the system. The complete equation is as follows:

$$Assig_i = Round[(Prio_i * Req_i * nCPU) / \Sigma\, Prio_j * Req_j]$$

As commented is Section 2.3.4, performance isolation of web applications is a concern in hosting platforms where applications share resources, because when an application overloads it can affect the performance of other applications. Consequently, it is a responsibility of the hosting platform to provide performance isolation. The ECM considers this issue when allocating processors to the applications. The ECM not only decides how many processors to assign to each application, but decides also which processors to assign to each application. In order to accomplish this, the ECM configures the CPU affinity mask of each application (using the Linux `sched_setaffinity` function) so that the application can run only in their assigned processors, and no other application can run in these processors, guarantying in this way performance isolation.

It is also desirable that ECM maximizes resource utilization in the hosting platform. In order to accomplish this, the ECM can decide under certain conditions that two applications share a given processor, trying to minimize impact on performance isolation. Current ECM implementation will decide to share a processor from application A to application B if the processor distribution policy has assigned to application A all the processors it requested and the number of processors assigned to application B is lower than the number it requested. Notice that, in this situation, it is possible that a fraction of a processor allocated to application A is not used, for example, if application A determines that it needs 2.5 processors, its processor request will be 3, thus a 0.5 processor may be not used.

The ECM has another feature very valuable in hosting platforms that earn money from applications depending on their resource usage. In these situations, hosting platforms need to know exactly how many resources have been used by each application. The ECM can easily provide this information, because it performs a complete accounting of all the resource allocations decided.

5.3.2 Evaluation

This section presents the evaluation results for the overload control approach including dynamic resource provisioning proposed in this thesis. The evaluation is divided in two parts. First, the accuracy of the mechanism for estimating the processor requirements of the application server is evaluated by comparing the execution of a single self-adaptive instance of the Tomcat server with this mechanism incorporated (self-managed Tomcat server) with respect to the original Tomcat. Second, the proposal combining dynamic resource provisioning and admission control is evaluated by running several experiments with two self-adaptive Tomcat instances in a multiprocessor hosting platform with the ECM.

Figure 5.21 shows the number of processors allocated to Tomcat comparing the original Tomcat server with respect to the self-managed Tomcat server. When running the original Tomcat, the hosting platform must perform static processor provisioning because it has no information about its processor requirements. If maximum application performance wants to be achieved, the hosting platform must allocate the maximum number of processors (four in this case) to the server. However, this provokes poor processor utilization when the original Tomcat requires fewer processors. On the other side, self-managed Tomcat is able to calculate accurately its processor requirements and communicate them to the hosting platform, which can allocate to the server only the required processors, as shown in Figure 5.21, avoiding processor under-utilization but ensuring performance.

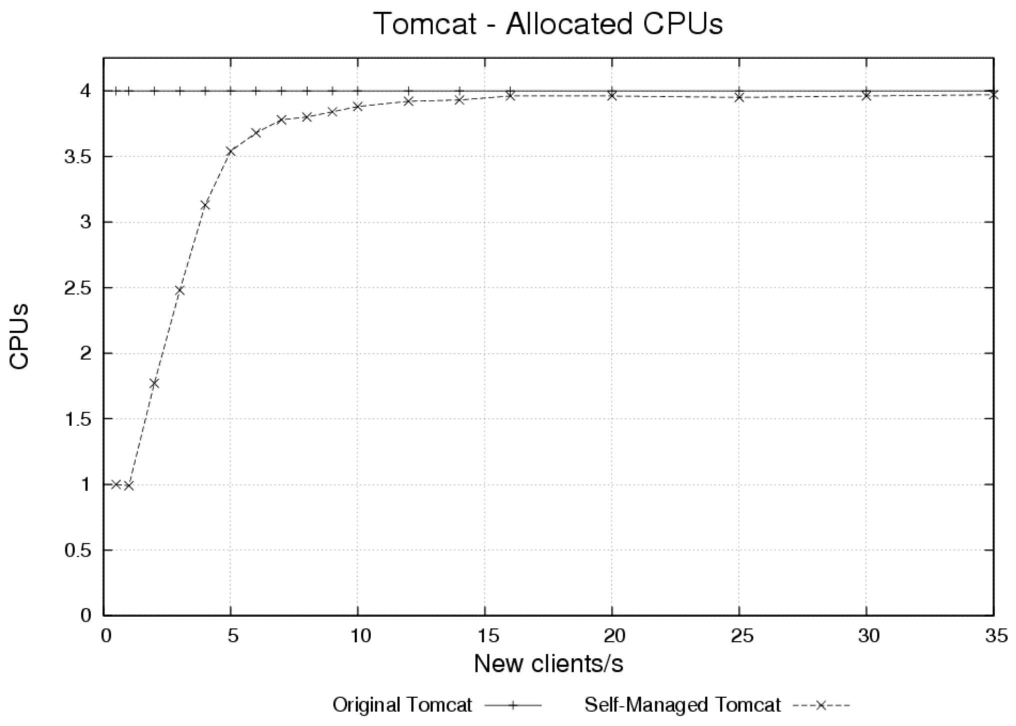


Figure 5.21. Original Tomcat vs. self-managed Tomcat number of allocated processors

The first multiprogrammed experiment consists of two Tomcat instances with the same priority running in a 4-way hosting platform. Each Tomcat instance has variable input load along time, which is shown in the top part of Figure 5.22. Input load distribution has been chosen in order to represent the different processor requirement combinations when running two Tomcat instances in a hosting platform. For example, between 0s and 1200s and between 2400s and 3000s the hosting platform can satisfy the processor requirements of the two Tomcat instances; this means that the hosting platform is not overloaded. In the other areas, the two Tomcat instances requirements exceed the number of processors of the hosting platform, thus the hosting platform is overloaded. In this case, some policy for processor distribution between the applications is required.

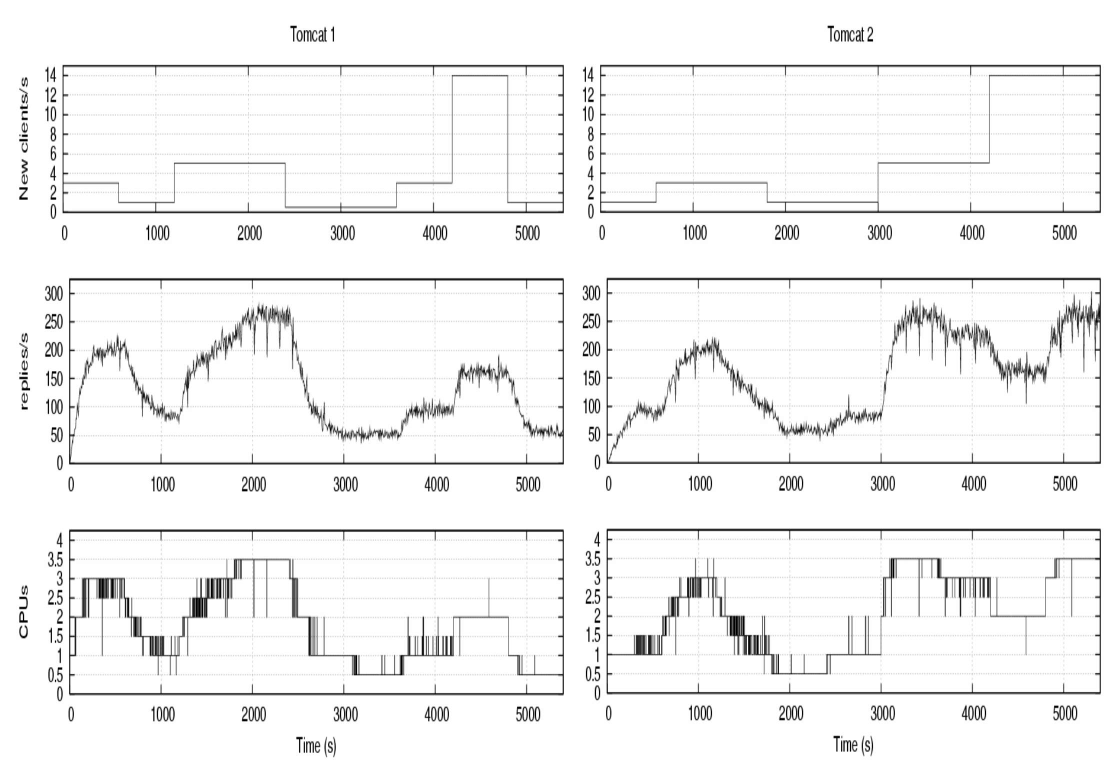


Figure 5.22. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances with the same priority

As well as the input load along time, Figure 5.22 also shows the processors allocation for each Tomcat instance (bottom part) and the throughput achieved with these processors allocations (middle part), presenting this information in a way that eases the correlation of the different metrics. Notice that, when the hosting platform is not overloaded, the two Tomcat instances receive all the processors they have requested, obtaining the corresponding throughput. When the hosting platform is overloaded, as the two instances have the same priority, the ECM distributes the available processors depending only on each Tomcat requirements, which depend on each Tomcat input load. Therefore, the Tomcat instance with higher input load (that is, with more processor requirements) is receiving more processors and hence achieving higher throughput. For example, between 1800s and 2400s, 5 new clients per second arrive to Tomcat 1 while to Tomcat 2 arrives only 1 new client per second. In this case, input load from Tomcat 1 is higher than input load from Tomcat 2, thus

Tomcat 1 will receive more processors than Tomcat 2. In particular, Tomcat 1 receives 3.5 processors on average (achieving a throughput around 260 replies/s) while Tomcat 2 receives only 0.5 processors on average (achieving a throughput around 50 replies/s). Notice that a processor is shared between Tomcat 1 and Tomcat 2. In the same way, between 3600s and 4200s, 5 new clients per second arrive to Tomcat 2 while to Tomcat 1 arrive only 3 new clients per second. In this case, input load from Tomcat 2 is higher than input load from Tomcat 1, thus Tomcat 2 will receive more processors than Tomcat 1. In particular, Tomcat 2 receives 3 processors on average (achieving a throughput around 230 replies/s) while Tomcat 1 receives only 1 processor on average (achieving a throughput around 50 replies/s). Finally, when the input load is the same for Tomcat 1 and Tomcat 2 (for instance between 4200s and 4800s), the two instances receive the same number of processors (two in this case), obtaining the same throughput (around 150 replies/s). In any case, as demonstrated in Chapter 4, the overload control mechanism ensures that although the number of required processors is not supplied, the QoS of admitted requests is maintained.

The second multiprogrammed experiment has the same configuration that the previous one but, in this case, Tomcat 1 has higher priority than Tomcat 2 (2 versus 1). As the two instances have different priority, the ECM distributes the available processors depending on each Tomcat requirements and on its priority, following the equation presented in Section 5.3.1.2. Figure 5.23 shows the results obtained for this experiment presenting these results in the same way as Figure 5.22. Notice that now between 1800s and 2400s, processors allocated to Tomcat 1 have increased, oscillating between 3.5 and 4 on average while processors allocated to Tomcat 2 have decreased, oscillating between 0 and 0.5 on average, because as well as having higher input load, Tomcat 1 has also higher priority than Tomcat 2. In the same way, between 3600s and 4200s, processors allocated to Tomcat 2 have decreased, oscillating between 2 and 2.5 on average while processors allocated to Tomcat 1 have increased, oscillating between 1.5 and 2 on average, because although Tomcat 2 has higher input load, Tomcat 1 has higher priority than Tomcat 2. Finally, between 4200s and 4800s, although the input load is the same for Tomcat 1 and Tomcat 2, Tomcat 1 receives now more processors than Tomcat 2 (3 versus 1), because Tomcat 1 has higher priority than Tomcat 2. With this processor allocation, Tomcat 1 obtains higher throughput than Tomcat 2 (around 230 replies/s versus 100 replies/s).

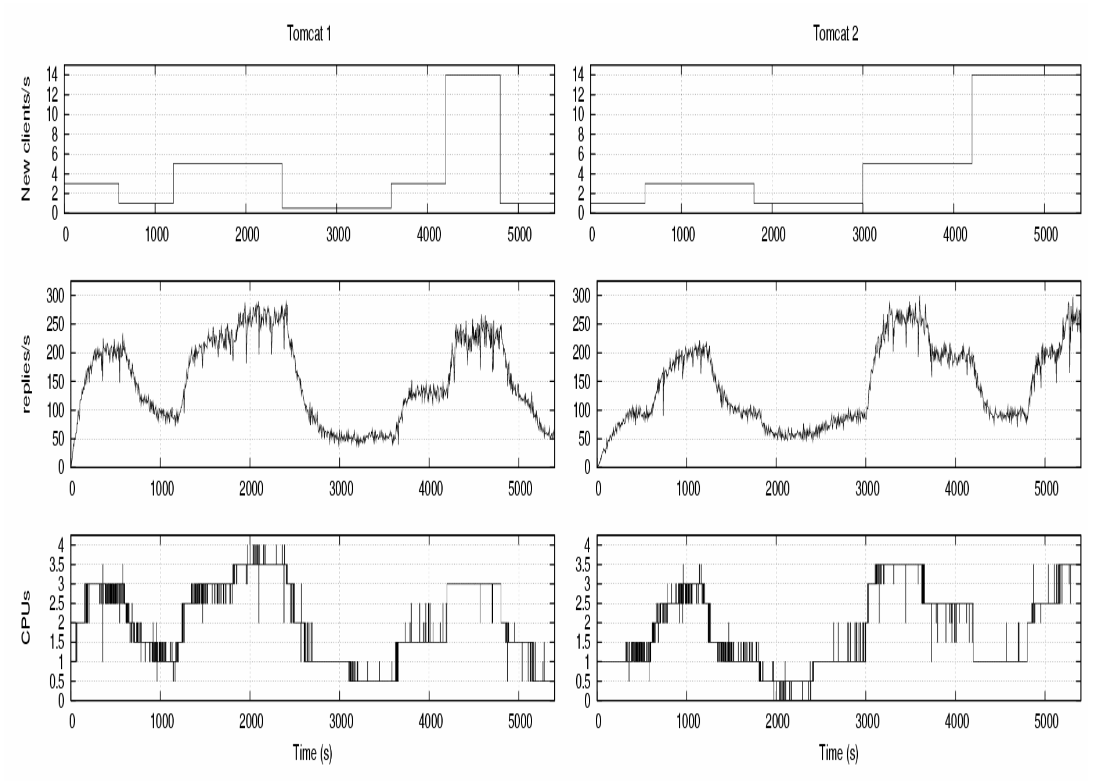


Figure 5.23. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances if Tomcat 1 has higher priority than Tomcat 2

The last multiprogrammed experiment has the same configuration that in the previous one, but with a slightly different behavior of the ECM in order to benefit the execution of high priority applications. In this experiment, a processor can be only shared from low priority applications to high priority applications, but not on the other side. Figure 5.24 shows the results obtained for this experiment presenting these results in the same way as Figure 5.22. As shown in this figure, between 1800s and 2400s, processors allocated to Tomcat 1 have increased to almost 4 on average while processors allocated to Tomcat 2 are now nearly 0, because Tomcat 1 has higher priority than Tomcat 2 and does not share processors. In the same way, between 3600s and 4200s, processors allocated to Tomcat 2 have decreased to 1 on average while processors allocated to Tomcat 1 have increased to 3 on average, because although Tomcat 2 has higher input load, Tomcat 1 has higher priority than Tomcat 2 and does not share processors. With this processor allocation, Tomcat 1 obtains now higher throughput than in the previous experiment (around 200 replies/s versus 130 replies/s) while Tomcat 2 achieves now lower throughput (around 100 replies/s versus 200 replies/s).

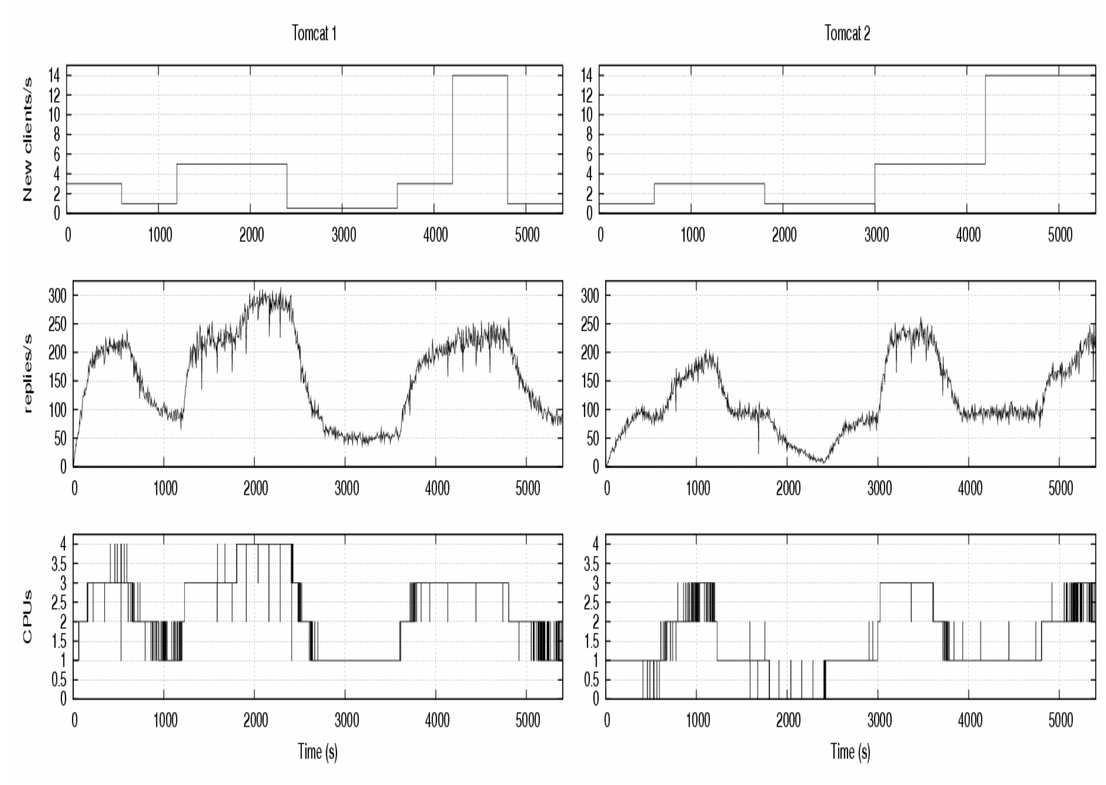


Figure 5.24. Incoming workload (top), achieved throughput (middle) and allocated processors (bottom) of two Tomcat instances if Tomcat 1 has higher priority than Tomcat 2 and Tomcat 1 does not share processors with Tomcat 2

5.4 Conclusions

The "Resource Provisioning for Multithreaded Java Applications" work area described in this chapter shows how, in addition to implement self-adaptive applications that can adapt their behavior depending on the available resources, the cooperation between the applications and the execution environment in order to manage efficiently the resources improves the performance of multithreaded Java applications on multiprogrammed shared-memory multiprocessors.

This thesis proposes the implementation of this cooperation based on establishing a bi-directional communication path between the applications and the underlying system. On one side, the applications request to the execution environment the number of processors they need. On the other side, the execution environment can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications, which are self-adaptive, can adapt their behavior to the assigned resources as described in Chapter 4.

This thesis contributes with the implementation of the cooperation between the execution environment and the applications for manage the resources as in HPC environments as in e-business environments. The implementation for HPC environments considers two different scenarios. In the first one, the application is able to inform the execution environment about its concurrency level using a service provided by the underlying thread library. As shown in the experimental results, the effect on performance of this communication is low when executing applications that create threads with a long lifetime. In the second scenario, in addition to this communication path, the execution environment is also able to inform the application about the resource provisioning decisions. As the application is malleable (i.e. self-adaptive), it is able to react to these decisions by changing the degree of parallelism that it is actually exploited from the application.

The experimental results show a noticeable impact on the final performance for malleable applications. Improvements avoiding performance degradation in non-overloaded multiprogrammed environments range from 7% to 31% when malleable applications do not adapt to the assigned processors, and from 12% to 33% otherwise. On multiprogrammed overloaded environments, improvements range from 10% to 26% when malleable applications do not adapt to the assigned processors, and from 8% to 58% otherwise. Notice that, in an overloaded system it is very important if applications are malleable, because there are not enough resources to satisfy all the requests. Although this scenario is based on malleable applications, this chapter has demonstrated that is also possible to maintain the efficiency of non-malleable applications. The performance degradation for this kind of applications is almost the same when running with Irix or with JNE.

The implementation of the cooperation between the execution environment and the applications for manage efficiently the resources in e-business environments uses an overload control approach for self-adaptive Java application servers running secure e-commerce applications that brings together admission control based on SSL connections differentiation and dynamic provisioning of platform resources in order to adapt to changing workloads avoiding the QoS degradation.

The overload control approach is based on a global resource manager responsible of distributing periodically the available processors among web applications following a determined policy. The resource manager can be configured to implement different policies, considering traditional indicators (i.e. response time) as well as e-business indicators (i.e. customer's priority). The resource manager and the applications cooperate to manage the resources using a bi-directional communication. On one side, the applications request to the resource manager the number of processors needed to handle their incoming load without QoS degradation. On the other side, the resource manager can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications can apply the admission control mechanism described in Chapter 4 that limits the number of admitted requests so they can be served with the allocated processors without degrading their QoS.

The experimental results demonstrate the benefit of combining dynamic resource provisioning and admission control to prevent overload of Java application servers in secure environments. Dynamic resource provisioning allows meeting the requirements of the application servers on demand and adapting to their changing resource needs. In this way, better resource utilization by extracting multiplexing gains can be achieved (resources not used by some application may be distributed among other applications) and the system can react to unexpected workload increases. On the other side, admission control based on SSL differentiation allows maintaining the response times in levels that guarantee good QoS and avoiding server throughput degradation (throughput degrades until approximately the 20% of the maximum achievable throughput when server overloads), while maximizing the number of sessions completed successfully.

The research performed in this work area has resulted in the following publications, including one journal, two international conferences (one submitted but not yet accepted) and one international workshop:

- J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. Dynamic Resource Provisioning for Self-Managed QoS-Aware Secure e-Commerce Applications in SMP Hosting Platforms. To be submitted to the 20th International Parallel and Distributed Symposium (IPDPS'06), Rhodes Island, Greece. April 26-29, 2006.

- J. Guitart, X. Martorell, J. Torres and E. Ayguadé. Application/Kernel Cooperation Towards the Efficient Execution of Shared-memory Parallel Java Codes. 17th International Parallel and Distributed Symposium (IPDPS'03), Nice, France. April 22-26, 2003.

- J. Guitart, X. Martorell, J. Torres and E. Ayguadé. Efficient Execution of Parallel Java Applications. 3rd Annual Workshop on Java for High Performance Computing (part of the 15th ACM International Conference on Supercomputing ICS'01), pp. 31-35, Sorrento, Italy. June 17, 2001.

- J. Oliver, E. Ayguadé, N. Navarro, J. Guitart and J. Torres. Strategies for Efficient Exploitation of Loop-level Parallelism in Java. Concurrency and Computation: Practice and Experience (Java Grande 2000 Special Issue), Vol.13 (8-9), pp. 663-680. ISSN 1532-0634, July 2001.

CHAPTER 6
RELATED WORK

6.1 Analysis and Visualization of Multithreaded Java Applications

Although a number of tools have been developed to monitor and analyze the performance of Java applications, only some of them target multithreaded Java applications, and none of them allow a fine-grain analysis of the applications behavior considering all levels involved in the application execution. Different approaches are used to carry on the instrumentation process. Paradyn [152] is a non-trace based tool that considers Java multithreaded applications and allows users to insert and remove instrumentation probes during program execution by dynamically relocating the code and adding pre and post instrumentation code. Jinsight [117], JaViz [91] and DejaVu [42] work with traces generated by an instrumented JVM. Jinsight and DejaVu allow the instrumentation of multithreaded Java applications while JaViz allows the instrumentation of client/server Java applications that use RMI. Other works allow the analysis of multithreaded Java applications by instrumenting the Java source code [16], thus requiring the recompilation of the application.

There is another set of proposals, such as Hprof (which is shipped with the standard Java SDK), TAU [127] and OptimizeIt [114], which offer maximum portability by using the Java Virtual Machine Profiler Interface [143] (JVMPI). JVMPI is an interface that profilers can use to obtain profiling information generated from de JVM. This means that all standard JVM is really an instrumented JVM that generates profiling information that can be captured using the JVMPI. With Hprof, all the information generated by the JVMPI can be accessed, directly or using some post-processing tool as PerfAnal [105] or Heap Analysis Tool [81] (HAT). OptimizeIt can be integrated with popular J2EE application servers. TAU allows the analysis of parallel Java applications based on MPI using visualizers as Racy and Vampir [115]. However, all these JVMPI-based tools suffer of large overheads due to the use of JVMPI.

Related work includes also other tools for the analysis and visualization of multithreaded applications, but these tools do not consider Java applications. For example, Sun Workshop Thread Event Analyzer [151] is based on the post-mortem analysis of traces obtained through shared libraries interposition; Socrates [145] allows the post-mortem analysis of traces obtained by instrumenting the application source code; Tmon [86] is a trace-based tool that obtains the profiling information by instrumenting the user threads library; and finally Gthread [153] is a trace-based tool that adds instrumentation information using macros that replace Pthreads library calls.

Finally, a number of tools have been developed specifically, or consider in any way the analysis of web applications performance. Some of these tools are, for instance, Wily Technology Solutions for Enterprise Java Application Management (Introscope) [149], Quest Software Solutions for Java/J2EE (JProbe, Performasure) [123] and Empirix Solutions for Web Application Performance (e-TEST, OneSight) [51].

All the tools commented report different metrics that measure and breakdown, in some way, the application performance. However, none of them enables a fine-grain analysis of the multithreaded execution and the scheduling issues involved in the execution of the threads that come from the Java application. This analysis requires different kind of information, which must be acquired at several levels, from the application to the system level.

Some tools focus the analysis on the application level (and the application server level, if applicable), neglecting the interaction with the system. Other tools incorporate the analysis of the system activity to their monitoring solution, but summarize this analysis giving general metrics (as CPU utilization or JVM memory usage) providing only a quantitative analysis of the server execution. Summarizing, existing tools base their analysis on calculating general metrics that intend to represent the system status. Although this information can be useful for the detection of some problems, it is often not sufficiently fine grained and lacks of flexibility. For this reason, this thesis proposes an analysis environment to perform a complete analysis of the applications behavior based on providing to the user detailed and correlated information about all levels involved in the application execution, giving him the chance to construct his own metrics, oriented to the kind of analysis he wants to perform.

6.2 Characterization of Java Application Servers Scalability

Application server scalability constitutes an important issue to support the increasing number of users of secure dynamic web sites. Although this thesis focuses on maintaining server scalability when running in secure environments adding more resources (vertical scaling), the large computational demand of SSL protocol can be handled using other approaches.

Major J2EE vendors such as BEA [17] or IBM [5][41] use clustering (horizontal scaling) to achieve scalability and high availability. Several studies evaluating server scalability using clustering have been performed [5][77], but none of them considers security issues.

Scalability can be also achieved delegating the security issues on a web server (e.g. Apache web server [9]) while the application server only processes dynamic web requests. In this case, the computational demand will be transferred to the web server, which can be optimized for SSL management.

It is also possible to add new specialized hardware for processing SSL requests [108], reducing the processor demand, but increasing the cost of the system.

Related with the vertical scalability covered in this thesis, some works have evaluated this scalability on web servers or application servers. For example, [18] and [79] only consider static web content, and in [8][18][79][98] the evaluation is limited to a numerical study without performing an analysis to justify the scalability results obtained. Besides, none of these works evaluates the effect of security on application server scalability.

Other works try 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 some parameters, including different JVM implementations, JVM flags and XML implementations has been studied in [96]. In the same way, the application server scalability using different mechanisms for generating dynamic web content has been evaluated in [32]. However, none of these works considers any kind of scalability relative to resources (neither vertical nor horizontal), neither the influence of security on the application server scalability.

Certain kind of analysis has been performed in some works. For example, [4] and [32] provide a quantitative analysis based on general metrics of the 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 when dealing with more sophisticated performance problems.

The influence of security on application server scalability has been covered in some works. For example, the performance and architectural impact of SSL on the servers in terms of various parameters such as throughput, utilization, cache sizes and cache miss ratios has been analyzed in [90], concluding that SSL increases computational cost of transactions by a factor of 5-7. The impact of each individual operation of TLS protocol in the context of web servers has been studied in [43], showing that key exchange is the slowest operation in the protocol. [59] analyzes the impact of full handshake in connection establishment and proposes caching sessions to reduce it.

Security for Web Services can be also provided with SSL, but other proposals as WS-Security [83], which uses industry standards like XML Encryption and XML Signature, have been made. Coupled with WS-SecureConversation, the advantage WS-Security has over SSL over HTTP is twofold: first, it works independently of the underlying transport protocol and second, it provides security mechanisms that operate in end-to-end scenarios (across trust boundaries) as opposed to point-to-point scenarios (i.e. SSL). Anyway, WS-Security requires also a large computational demand to support the encryption mechanisms, making most of the conclusions obtained in this thesis valid in Web Services environments too.

This thesis intends to achieve a complete characterization of dynamic web applications using SSL vertical scalability determining the causes of server overload performing a detailed analysis of application server behavior considering all levels involved in the execution of dynamic web applications.

6.3 Overload Control and Resource Provisioning in Web Environments

The effect of overload on web applications has been covered in several works, applying different perspectives in order to prevent these effects. These different approaches can be resumed on request scheduling, admission control, service differentiation, service degradation, resource management and almost any combination of them.

Request scheduling refers to the order in which concurrent requests should be served. Typically, servers have been left this ordination to the operating system. But, as it is well know from queuing theory that shortest remaining processing time first (SRPT) scheduling minimizes queuing time (and therefore the average response time), some proposals [46][80] implement policies based on this algorithm to prioritize the service of short static content requests in front of long requests. This prioritized scheduling in web servers has been proven effective in providing significantly better response time to high priority requests at relatively low cost to lower priority requests. Although scheduling can improve response times, under extreme overloads other mechanisms become indispensable. Anyway, better scheduling can always be complementary to any other mechanism.

Admission control is based on reducing the amount of work the server accepts when it is faced with overload. Service differentiation is based on differentiating classes of customers so that response times of preferred clients do not suffer in the presence of overload. Admission control and service differentiation have been combined in some works to prevent server overload. For example, [144] presents three kernel-based mechanisms that include restricting incoming SYN packets to control TCP connection rate, prioritized listen queue and HTTP header-based classification providing service differentiation. ACES [38] attempts to limit the number of admitted requests based on estimated service times, allowing also service prioritization. The evaluation of this approach is done based only on simulation. Other works have considered dynamic web content. An adaptive approach to overload control in the context of the SEDA [148] Web server is described in [147]. SEDA decomposes services into multiple stages, each one of which can perform admission control based on monitoring the response time through the stage. The evaluation includes dynamic content in the form of a web-based email service. In [50], the authors present an admission control mechanism for e-commerce sites that externally observes execution costs of requests, distinguishing different requests types. Yaksha [89] implements a self-tuning proportional integral controller for admission control in multi-tier e-commerce applications using a single queue model.

Service degradation is based on avoiding refusing clients as a response to overload but reducing the service offered to clients [1][37][140][147], for example in the form on providing smaller content (e.g. lower resolution images).

Recent studies [7][35][36] have reported the considerable benefit of dynamically adjusting resource allocations to handle variable workloads. This premise has motivated the proposal of several techniques to dynamically provision resources to applications in on demand hosting platforms. Depending on the mechanism used to decide the resource allocations, these proposals can be classified on: control theoretic approaches with a feedback element [2], open-loop approaches based on queuing models to achieve resource guarantees [34][48][97] and observation-based approaches that use runtime measurements to compute the relationship between resources and QoS goal [122]. Control theory solutions require training the system at different operating points to determine the control parameters for a given workload. Queuing models are useful for steady state analysis but do not handle transients accurately. Observation-based approaches are most suited for handling varying workloads and non-linear behaviors. Depending on the hosting platform architecture considered, resource management in a single machine has been covered in [12], proposing resource containers as an operating system abstraction that embodies a resource. The problem of provisioning resources in cluster architectures has been addressed in [10][124] by allocating entire machines (dedicated model) and in [34][122][141] by sharing node resources among multiple applications (shared model).

Cataclysm [140] performs overload control bringing together admission control, adaptive service degradation and dynamic provisioning of platform resources, demonstrating that the most effective way to handle overload must consider the combination of techniques. In this aspect, this work is similar to the proposal in this thesis.

On most of the prior work, overload control is performed on per request basis, which may not be adequate for many session-based applications, such as e-commerce applications. A session-based admission control scheme has been reported in [40]. This approach allows sessions to run to completion even under overload, denying all access when the server load exceeds a predefined threshold. Another approach to session-based admission control based on the characterization of a commercial web server log, which discriminates the scheduling of requests based on the probability of completion of the session that the requests belong to, is presented in [39].

The overload control mechanism proposed in this thesis combines important aspects that previous work has considered in isolation or simply has ignored. First, it considers dynamic web content instead of simpler static web content. Second, it focuses on session-based applications considering the particularities of these applications when performing admission control. Third, it combines several techniques as admission control, service differentiation and dynamic resource provisioning that have been demonstrated to be useful to prevent overload [140] instead of considering each technique in isolation. Fourth, this mechanism is fully adaptive to the available resources and to the number of connections in the server instead of using predefined thresholds. Fifth, the resource provisioning mechanism incorporates e-business indicators instead of only considering conventional performance metrics such as response time and throughput. Finally, it considers overload control on secure web applications while none of the above works has covered this issue.

6.4 Resource Provisioning in HPC Environments

Experience on real systems shows that with contemporary kernel schedulers, parallel applications suffer from performance degradation when executed in an open multiprogrammed environment. As a consequence, intervention from the system administrator is usually required, in order to guarantee a minimum quality of service with respect to the resources allocated to each parallel application (CPU time, memory etc.). Although the use of sophisticated queuing systems and system administration policies (HP-UX Workload Manager [130], IBM AIX WLM [82], Solaris RM [138], IRIX Miser Batch Processing System [128], etc.) may improve the execution conditions for parallel applications, the use of hard limits for the execution of parallel jobs with queuing systems may jeopardize global system performance in terms of utilization and fairness.

Even with convenient queuing systems and system administrator's policies, application and system performance may still suffer because users are only able to provide very coarse descriptions of the resource requirements of their jobs (number of processors, CPU time, etc.). Fine grain events that happen at execution time (spawning parallelism, sequential code, synchronizations, etc.), which are very important for performance, can only be handled at the level of the runtime system, through an efficient cooperation interface with the operating system. This scenario assumes applications that are able to adapt their behavior to the amount of resources allocated to them. This information is obtained by establishing a dialog with the execution environment.

Several proposals of cooperation between the execution environment and the applications appear in the related work, but none of them consider multithreaded Java applications. For example, Process Control [139] proposes to share a counter of running processes, but the concurrency level of an application is inferred by the execution environment instead of being specified by the application. Process Control, Scheduler Activations [6] and First-Class Threads [99] use signals or upcalls to inform the user level about preemptions.

The Nanos RM [100] (NRM) is an application-oriented resource manager, i.e. the unit of resource allocation and management is the parallel application. Other resource managers, such as the Solaris RM or the AIX WLM, work at workload or user granularity. Having parallel applications as units for resource management allows the application of performance-driven policies [45] that take into account the characteristics of these applications (e.g. speedup or efficiency in the use of resources). The NRM takes decisions at the same level than the kernel does. This means that it does not only allocate processors to a particular application, but also it performs the mapping between kernel threads and processors and controls the initial memory placement. This is an issue that is important to consider in the Java environment using the native threads model (several kernel threads in contraposition to the green threads model that just uses one kernel thread for all the Java threads in the application).

The Jikes RVM [3] implements a different thread model. It provides virtual processors in the Java runtime system to execute the Java threads. Usually, there are more Java threads than virtual processors. Each virtual processor is scheduled onto a pthread. This means that, as the other threads models do, Jikes relies on the Pthreads library for scheduling the pthreads over the kernel threads offered by the operating system, suffering the same performance degradation problems for parallel Java applications. Therefore, Jikes can also benefit of the solutions proposed in this thesis.

CHAPTER 7
CONCLUSIONS

7.1 Conclusions

This thesis has contributed in the resolution of the performance problems faced when using the Java language in parallel environments (from HPC environments to e-business environments). The contributions have included the definition of an environment to analyze and understand the behavior of multithreaded Java applications. The main contribution of this environment is that all levels in the execution (application, application server, JVM and operating system) are correlated. This is very important to understand how this kind of applications behaves when executed on execution environments that include servers and virtual machines. In addition, and based on the understanding gathered using the proposed analysis environment, this thesis has performed research on scheduling mechanisms and policies oriented towards the efficient execution of multithreaded Java applications on multiprocessor systems considering the interactions and coordination between scheduling mechanisms and policies at different levels: application, application server, JVM, threads library and operating system.

In order to achieve these main objectives, the thesis has been divided in the following work areas.

- Analysis and Visualization of Multithreaded Java Applications
- Self-Adaptive Multithreaded Java Applications
- Resource Provisioning for Multithreaded Java Applications

7.1.1 Analysis and Visualization of Multithreaded Java Applications

The "Analysis and Visualization of Multithreaded Java Applications" work area claims that a real performance improvement on multithreaded Java applications must be preceded by a fine-grain analysis of applications behavior, considering all levels involved in the applications execution, in order to detect the bottlenecks for performance.

Therefore, the main contribution in this work area has been the proposal of a performance analysis framework to perform a complete analysis of the Java applications behavior based on providing to the user detailed information about all levels involved in the application execution, giving him the chance to construct his own metrics, oriented to the kind of analysis he wants to perform.

The proposed performance analysis framework consists of two tools: an instrumentation tool, called JIS (Java Instrumentation Suite), and an analysis and visualization tool, called Paraver. When instrumenting a given application, JIS generates a trace in which the information collected from all levels has been correlated and merged. Later, the trace can be visualized and analyzed with Paraver (qualitatively and quantitatively) to identify the performance bottlenecks of the application.

JIS provides information from all levels involved in the application execution. From the system level, information about threads state and system calls (I/O, sockets, memory management and thread management) can be obtained. Several implementations have been performed depending on the underlying platform. A dynamic interposition mechanism that obtains information about the supporting threads layer (i.e. Pthreads library) without recompilation has been implemented for the SGI Irix platform. In the same way, a device driver that gets information from a patched Linux kernel has been developed for the Linux platform. JIS uses the JVMPI to obtain information from the JVM level. At this level of analysis, the user can obtain information about several Java abstractions like classes, objects, methods, threads and monitors, but JIS only obtains at this level the name of the Java threads and information from the different Java Monitors (when they are entered, exited or contended), due to the large overhead produced when using JVMPI. Information relative to services (i.e. servlets and EJB), requests, connections or transactions can be obtained from the application server level. Moreover, some extra information can be added to the final trace file by generating user events from the application code. Information at these levels can be inserted by hard-coding JNI calls to the instrumentation library on the server or the application source or by introducing them dynamically using Aspect programming techniques without source code recompilation.

As a special case of instrumentation at the application level, support for JOMP applications has been added to JIS. JOMP includes OpenMP-like extensions to specify parallelism in Java applications using a shared-memory programming paradigm. This instrumentation approach has been designed to provide a detailed analysis of the parallel behavior at the JOMP programming model level. At this level, the user is faced with parallel, work-sharing and synchronization constructs. The JOMP compiler has been modified to inject JNI calls to the instrumentation library during the code generation phase at specific points in the source code.

The experience in this work area has demonstrated the benefit of disposing of correlated information about all the levels involved in Java applications execution to perform a fine-grain analysis of their behavior. This thesis claims that a real performance improvement on multithreaded Java applications execution can only be achieved if the performance bottlenecks at all levels can be identified.

7.1.2 Self-Adaptive Multithreaded Java Applications

The "Self-Adaptive Multithreaded Java Applications" work area has demonstrated the benefit of implementing self-adaptive multithreaded Java applications in order to achieve good performance when using Java in parallel environments. Self-adaptive applications are those applications that can adapt their behavior to the amount of resources allocated to them.

This thesis has presented two contributions in this work area towards achieving self-adaptive applications and has demonstrated the performance improvement obtained when having this kind of applications. The first contribution in this work area has been a complete characterization of the scalability of Java application servers when executing secure dynamic web applications. This characterization is divided in two parts:

The first part has consisted of measuring Tomcat vertical scalability (i.e. adding more processors) when using SSL and analyzing the effect of this addition on server scalability. The results have confirmed that running with more processors makes the server able to handle more clients before overloading and even when the server has reached an overloaded state, better throughput can be obtained if running with more processors. The second part has involved an analysis of the causes of server overload when running with different number of processors using the performance analysis framework proposed in Chapter 3 of this thesis. The analysis has revealed that the processor is a bottleneck for Tomcat performance on secure environments (the massive arrival of new SSL connections demands a computational power that the system is unable to supply and the performance is degraded) and could make sense to upgrade the system adding more processors to improve the server scalability. The analysis results also have demonstrated the convenience of incorporating to the Tomcat server some kind of overload control mechanism to avoid the throughput degradation produced due to the massive arrival of new SSL connections that the analysis has detected.

Based on the conclusions extracted from this analysis, the second contribution has been the implementation of a session-based adaptive overload control mechanism based on SSL connections differentiation and admission control. SSL connections differentiation has been accomplished using a possible extension of the JSSE package in order to allow distinguishing resumed SSL connections (that reuse an existing SSL session on server) from new SSL connections. This feature has been used to implement a session-based adaptive admission control mechanism that has been incorporated to the Tomcat server. This admission control mechanism differentiates new SSL connections from resumed SSL connections limiting the acceptation of new SSL connections to the maximum number acceptable with the available resources without overloading the server, while accepting all the resumed SSL connections in order to maximize the number of sessions completed successfully, allowing to e-commerce sites based on SSL to increase the number of transactions completed.

The experimental results demonstrate that the proposed mechanism prevents the overload of Java application servers in secure environments. It maintains response time in levels that guarantee good QoS and avoids completely throughput degradation (throughput degrades until approximately the 20% of the maximum achievable throughput when server overloads), while maximizes the number of sessions completed successfully (which is a very important metric on e-commerce environments). These results confirm that security must be considered as an important issue that can heavily affect the scalability and performance of Java application servers.

7.1.3 Resource Provisioning for Multithreaded Java Applications

The “Resource Provisioning for Multithreaded Java Applications” work area has shown how, in addition to implement self-adaptive applications that can adapt their behavior depending on the available resources, the cooperation between the applications and the execution environment in order to manage efficiently the resources improves the performance of multithreaded Java applications on multiprogrammed shared-memory multiprocessors.

This thesis has proposed the implementation of this cooperation based on establishing a bi-directional communication path between the applications and the underlying system. On one side, the applications request to the execution environment the number of processors they need. On the other side, the execution environment can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications, which are self-adaptive, can adapt their behavior to the amount of resources allocated to them as described in Chapter 4.

This thesis has contributed with the implementation of the cooperation between the execution environment and the applications for manage the resources as in HPC environments as in e-business environments. The implementation for HPC environments considers two different scenarios. In the first one, the application is able to inform the execution environment about its concurrency level using a service provided by the underlying thread library. As shown in the experimental results, the effect on performance of this communication is low when executing applications that create threads with a long lifetime. In the second scenario, in addition to this communication path, the execution environment is also able to inform the application about the resource provisioning decisions. As the application is malleable (i.e. self-adaptive), it is able to react to these decisions by changing the degree of parallelism that it is actually exploited from the application.

The experimental results show a noticeable impact on the final performance for malleable applications. Improvements avoiding performance degradation in non-overloaded multiprogrammed environments range from 7% to 31% when malleable applications do not adapt to the assigned processors, and from 12% to 33% otherwise. On multiprogrammed overloaded environments, improvements range from 10% to 26% when malleable applications do not adapt to the assigned processors, and from

8% to 58% otherwise. Notice that, in an overloaded system it is very important if applications are malleable, because there are not enough resources to satisfy all the requests. Although this scenario is based on malleable applications, this chapter has demonstrated that is also possible to maintain the efficiency of non-malleable applications. The performance degradation for this kind of applications is almost the same when running with Irix or with JNE.

The implementation of the cooperation between the execution environment and the applications for manage efficiently the resources in e-business environments has used an overload control approach for self-adaptive Java application servers running secure e-commerce applications that brings together admission control based on SSL connections differentiation and dynamic provisioning of platform resources in order to adapt to changing workloads avoiding the QoS degradation.

The overload control approach is based on a global resource manager responsible of distributing periodically the available processors among web applications following a determined policy. The resource manager can be configured to implement different policies, considering traditional indicators (i.e. response time) as well as e-business indicators (i.e. customer's priority). The resource manager and the applications cooperate to manage the resources using a bi-directional communication. On one side, the applications request to the resource manager the number of processors needed to handle their incoming load without QoS degradation. On the other side, the resource manager can be requested at any time by the applications to inform them about their processor assignments. With this information, the applications can apply the admission control mechanism described in Chapter 4 that limits the number of admitted requests so they can be served with the allocated processors without degrading their QoS.

The experimental results have demonstrated the benefit of combining dynamic resource provisioning and admission control to prevent overload of Java application servers in secure environments. On one side, dynamic resource provisioning allows meeting the requirements of the application servers on demand and adapting to their changing resource needs. In this way, better resource utilization by extracting multiplexing gains can be achieved (resources not used by some application may be distributed among other applications) and the system can react to unexpected workload increases. On the other side, admission control based on SSL differentiation allows maintaining the response times in levels that guarantee good QoS and avoiding server throughput degradation (throughput degrades until approximately the 20% of the maximum achievable throughput when server overloads), while maximizing the number of sessions completed successfully.

The work performed in this thesis has resulted in several publications that support the quality of the contributions, including one journal, seven international conferences (one submitted but not yet accepted), two international workshops, three national conferences and ten technical reports.

7.2 Future Work

The work performed in this thesis opens several interesting ways that can be explored as a future work.

- This thesis has focused on self-adaptive application servers, i.e. servers that adapt their behavior to the amount of resources allocated by the system by limiting the incoming workload. However, in the way towards full "autonomic computing" it is desirable that these servers are also able to self-configure themselves, that is adjust dynamically some configuration parameters (e.g. the thread pool size) depending on the server workload and the system conditions in order to achieve the maximum performance and exploit efficiently the resources. These self-configuring capabilities can be achieved in the Tomcat server by using the JMX Proxy Servlet, which is a lightweight proxy that allows dynamically getting and setting the Tomcat internal configuration parameters.
- This thesis has considered e-business environments based on a single multiprocessor machine. However, today is common to find hosting platforms based on clusters of machines, each one running one o more applications. Future work may consider the extension of the proposed mechanisms to these architectures. In this scenario, the provisioning technique must determine how many nodes to allocate to each application and decide how to partition resources on each node among competing applications (if the node has been decided to be shared) depending on each application workload. A load balancer will be also necessary to distribute the incoming client requests into the different nodes. The load balancer will assign a client request to a node

chosen from the nodes assigned to the application the request belongs to, trying to balance the workload that the different nodes assigned to this application must face.

- The J2EE specification defines several types of components to create web applications, comprising Java Servlets (as considered in this thesis), Java Server Pages (JSP) and Enterprise Java Beans (EJB). The EJB are business components intended for the creation of complex and widely distributed web applications. These objectives are achieved at the cost of introducing a much higher level of complexity in the J2EE container. This additional complexity offers a great opportunity to propose new resource management mechanisms and policies, adapted to some of the especial requirements of an EJB container: EJB pools and caches, and persistence and transaction managers. The management strategies applied to an EJB container should cooperate with the system resource management techniques proposed in this thesis.
- Resource provisioning proposed in this thesis has focused on processors management, because the work is oriented towards secure e-business workloads, which are CPU-intensive. Of course, other kind of workloads will need an efficient management of other resources (for instance, network or database) to achieve good performance. The cooperation between the applications and the execution environment proposed in this thesis can be extended to consider these resources.
- This thesis has demonstrated the benefit of considering e-business indicators when designing policies for provisioning resources to the servers, using as an example a simple indicator: the customer’s priority. Future work may consider the implementation of more sophisticated policies using other e-business indicators of great interest for the e-commerce sites, such as the revenue generated. For instance, a policy could prioritize those requests belonging to sessions that are about to complete (for example, about to purchase a product), because those requests are likely to generate more revenue for the site.

APPENDICES

A. Java Grande Benchmarks

A.1 Section 1: Low Level Operations

➢ ForkJoin

This benchmark measures the time spent creating and joining threads. Performance is measured in fork-join operations per second.

➢ Barrier

This measures the performance of barrier synchronization. Performance is measured in barrier operations per second. Two types of barriers have been implemented. The first of these uses a shared counter. When a thread calls the barrier routine the counter is incremented. The thread then calls the `wait()` method. When the final thread enters the barrier, the counter is incremented and `notifyAll()` called, signaling all the other threads. The second of these is a static 4-way tournament barrier. This is a lock-free barrier, whose correctness cannot be formally guaranteed under the current, somewhat ambiguous, specification of the Java memory model. However, we have observed no such problems in practice. This barrier is used where barrier synchronization is required in Sections 2 and 3 of the suite.

➢ Sync

This benchmark measures the performance of synchronized methods and synchronized blocks. Performance is measured in synchronizations per second. The `Method` benchmark in the serial suite measures the performance of synchronized methods on a single thread. Here we measure the performance on multiple threads, where there is guaranteed to be contention for the object locks.

A.2 Section 2: Kernels

➢ **Crypt: IDEA encryption**

Crypt performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes. Performance units are bytes per second. It is bit/byte operation intensive. This algorithm involves two principle loops, whose iterations are independent and are divided between the threads in a block fashion.

Size	**N**
A	3,000,000
B	20,000,000
C	50,000,000

➢ **LUFact: LU factorization**

Solves an N x N linear system using LU factorization followed by a triangular solve. This is a Java version of the well-known Linpack benchmark. Performance units are Mflops per second. It is memory and floating point intensive. The factorization is the only part of the computation performed that is parallelized: the remainder is computed in serial. Iterations of the double loop over the trailing block of the matrix are independent and the work is divided between the threads in a block fashion. Barrier synchronization is required before and after the parallel loop.

Size	**N**
A	500
B	1,000
C	2,000

➢ **SOR: Successive over-relaxation**

The SOR benchmark performs 100 iterations of successive over-relaxation on an N x N grid. The performance reported is in iterations per second. This benchmark involves an outer loop over iterations and two inner loops, each looping over the grid. In order to update elements of the principle array during each iteration, neighboring elements of the array are required, including elements previously updated. Hence this benchmark is, in this form, inherently serial. To allow parallelization to be carried out the algorithm has been modified to use a “red-black” ordering mechanism. This allows the loop over array rows to be parallelized, hence the outer loop over elements has been distributed between threads in a block manner. Only nearest neighbor synchronization is required, rather than a full barrier.

Size	N
A	1,000
B	1,500
C	2,000

- **Series: Fourier coefficient analysis**

This benchmark computes the first N Fourier coefficients of the function `f(x) = (x+1)^x` on the interval 0,2. Performance units are coefficients per second. This benchmark heavily exercises transcendental and trigonometric functions. The most time consuming component of the benchmark is the loop over the Fourier coefficients. Each iteration of the loop is independent of every other loop and the work may be distributed simply between the threads. The work of this loop is divided evenly between the threads in a block fashion, with each thread responsible for updating the elements of its own block.

Size	N
A	10,000
B	100,000
C	1,000,000

- **Sparse: Sparse matrix multiplication**

This uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references. An N x N sparse matrix is used for 200 iterations. The principle computation involves an outer loop over iterations and an inner loop over the size of the principal arrays. The simplest parallelization mechanism is to divide the loop over the array length between threads. Parallelizing this loop creates the potential for more than one thread to up-date the same element of the result vector. To avoid this the non zero elements are sorted by their row value. The loop has then been parallelized by dividing the iterations into blocks, which are approximately equal, but adjusted to ensure that no row is access by more than one thread.

Size	N
A	50,000
B	100,000
C	500,000

A.3 Section 3: Large Scale Applications

- **MonteCarlo: Monte Carlo simulation**

A financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates N sample time series with the same mean and fluctuation as a series of historical data. Performance is measured in samples per second. The principle loop over number of Monte Carlo runs can be easily parallelized by dividing the work in a block fashion.

Size	N
A	2,000
B	60,000

- **RayTracer: 3D ray tracer**

This benchmark measures the performance of a 3D raytracer. The scene rendered contains 64 spheres, and is rendered at a resolution of N x N pixels. The performance is measured in pixels per second. The outermost loop (over rows of pixels) has been parallelized using a cyclic distribution for load balance. Since the scene data is fairly small, a copy of the scene is created for each thread. This allows optimizations in the serial code, principally the use of class variables for temporary storage, to be carried over to the parallel version.

Size	N
A	150
B	500

- **Euler: Computational fluid dynamics**

The Euler benchmark solves the time-dependent Euler equations for flow in a channel with a "bump" on one of the walls. A structured, irregular, N x 4N mesh is employed, and the solution method is a finite volume scheme using a fourth order

Runge-Kutta method with both second and fourth order damping. The solution is iterated for 200 timesteps. Performance is reported in units of timesteps per second.

Size	N
A	64
B	96

- **MolDyn: Molecular dynamics simulation**

MolDyn is an N-body code modeling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. Performance is reported in interactions per second. The number of particles is give by N. The original Fortran 77 code was written by Dieter Heerman, Institut für Theoretische Physik, Germany and converted to Java by Lorna Smith, EPCC. The computationally intense component of the benchmark is the force calculation, which calculates the force on a particle in a pair wise manner. This involves an outer loop over all particles in the system and an inner loop ranging from the current particle number to the total number of particles. The outer loop has been parallelized by dividing the range of the iterations of the outer loop between the threads, in a cyclic manner to avoid load imbalance. A copy of the data structure containing the force updates is created on each thread. Each thread accumulates force updates in its own copy. Once the force calculation is complete, these arrays are reduced to a single total force for each particle.

Size	N
A	2,048
B	8,788

BIBLIOGRAPHY

[1] T. Abdelzaher and N. Bhatti. Web Content Adaptation to Improve Server Overload Behavior. Computer Networks, Vol. 31 (11-16), pp. 1563-1577. May 1999.

[2] T. Abdelzaher, K. Shin and N. Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. IEEE Transactions on Parallel and Distributed Systems Vol. 13 (1), pp. 80-96. January 2002.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. IBM System Journal, Vol. 39 (1), 2000, pp. 211-238.

[4] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. IEEE 5th Annual Workshop on Workload Characterization (WWC-5), Austin, Texas, USA. November 25, 2002.

[5] Y. An, T. K. T. Lau and P. Shum. A Scalability Study for WebSphere Application Server and DB2. IBM white paper. January 2002. http://www-106.ibm.com/developerworks/db2/library/techarticle/0202an/0202an.pdf

[6] T. Anderson, B. Bershad, E. Lazowska and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. 13th ACM Symposium on Operating System Principles (SOSP'91), pp. 95-109, Pacific Grove, California, USA. October 13-16, 1991.

[7] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the Resource Savings of Utility Computing Models. Technical Report HPL-2002-339, HP Labs. December 2002.

[8] S. Anne, A. Dickson, D. Eaton, J. Guizan and R. Maiolini. JBoss 3.2.1 vs. WebSphere 5.0.2 Trade3 Benchmark. SMP Scaling: Comparison report. SWG Competitive Technology Lab. October 2003. http://www.werner.be/blog/resources/werner/JBoss_3.2.1_vs_WAS_5.0.2.pdf

[9] Apache HTTP Server Project
http://httpd.apache.org/

[10] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA-based Management of a Computing Utility. IFIP/IEEE Symposium on Integrated Network Management (IM 2001), pp. 855-868, Seattle, Washington, USA. May 14-18, 2001.

[11] AutoTune web site
http://www.research.ibm.com/PM/

[12] G. Banga, P. Druschel and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. 3rd Symposium on Operating Systems Design and Implementation (OSDI'99), pp. 45-58, New Orleans, Louisiana, USA. February 22-25, 1999.

[13] Barcelona eDragon Research Group
http://www.cepba.upc.es/eDragon

[14] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. ACM SIGMETRICS'98, pp. 151-160, Madison, Wisconsin, USA. June 24-26, 1998.

[15] J. Bartolomé and J. Guitart. A Survey on Java Profiling Tools. Research Report number: UPC-DAC-2001-13 / UPC-CEPBA-2001-10, April 2001.

[16] A. Bechini and C.A. Prete. Instrumentation of Concurrent Java Applications for Program Behavior Investigation. 1st Annual Workshop on Java for High Performance Computing (part of the 13th ACM International Conference on Supercomputing ICS'99), pp. 21-29, Rhodes, Greece. June 20, 1999.

[17] BEA Systems, Inc. Achieving Scalability and High Availability for E-Business. BEA white paper. March 2003. http://dev2dev.bea.com/products/wlserver81/whitepapers/WLS_81_Clustering.jsp

[18] V. Beltran, D. Carrera, J. Torres and E. Ayguade. Evaluating the Scalability of Java Event-Driven Web Servers. 2004 International Conference on Parallel Processing (ICPP'04), pp. 134-142, Montreal, Canada. August 15-18, 2004.

[19] V. Beltran, J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta. Performance Impact of Using SSL on Dynamic Web Applications. XV Jornadas de Paralelismo, pp. 471-476, Almeria, Spain. September 15-17, 2004.

[20] A.J.C. Bik and D.B. Gannon. Automatically Exploiting Implicit Parallelism in Java. Concurrency: Practice and Experience, Vol. 9 (6), pp.579-619. June 1997.

[21] A.J.C. Bik and D.B. Gannon. Javar: A Prototype Java Restructuring Compiler. UICS Technical Report TR487, July 1997.

[22] J.M. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. 1st European Workshop on OpenMP (EWOMP'99), pp. 99-105, Lund, Sweden. September 30 - October 1, 1999.

[23] J.M. Bull and M.E. Kambites. JOMP - an OpenMP-like Interface for Java. 2000 ACM Java Grande Conference, pp. 45-53, San Francisco, California, USA. June 3-5, 2000.

[24] J.M. Bull, L.A. Smith, L. Pottage and R. Freeman. Benchmarking Java against C and Fortran for Scientific Applications. ACM Java Grande/ISCOPE 2001 Conference, pp. 97-105, Stanford, California, USA. June 2-4, 2001.

[25] J.M. Bull, M.D. Westhead, M.E. Kambites and J.Obdrvzalek. Towards OpenMP for Java. 2nd European Workshop on OpenMP (EWOMP'00), pp. 98-105, Edimburgh, UK. September 14-15, 2000.

[26] B. Carpenter, G. Zhang, G. Fox, X. Li and Y. Wen. HPJava: Data Parallel Extensions to Java. Concurrency: Practice and Experience, Vol. 10 (11-13), pp. 873-877. September 1998.

[27] D. Carrera, J. Guitart, J. Bartolome, J. Torres and E. Ayguadé. JIS-JVMPI per Linux IA32: Instrumentació d'aplicacions Java en un entorn Linux. Research Report number: UPC-DAC-2002-36 / UPC-CEPBA-2002-13, July 2002.

[28] D. Carrera, J. Guitart, V. Beltran, J. Torres and E. Ayguadé. Performance Impact of the Grid Middleware. In Engineering the Grid: Status and Perspective, American Scientific Publishers, May 2005.

[29] D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. An Instrumentation Tool for Threaded Java Application Servers. XIII Jornadas de Paralelismo, pp. 205-210, Lleida, Spain. September 9-11, 2002.

[30] D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. Complete Instrumentation Requirements for Performance Analysis of Web based Technologies. 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), pp. 166-175, Austin, Texas, USA. March 6-8, 2003.

[31] D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. An Instrumentation Environment for Java Application Servers. Research Report number: UPC-DAC-2002-55 / UPC-CEPBA-2002-20, December 2002.

[32] E. Cecchet, J. Marguerite and W. Zwaenepoel. Performance and Scalability of EJB Applications. 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), pp. 246-261. Seattle, Washington, USA. November 4-8, 2002

[33] CEPBA web site
http://www.cepba.upc.edu/

[34] A. Chandra, W. Gong and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. 11th International Workshop on Quality of Service (IWQoS 2003), pp. 381-400, Berkeley, California, USA. June 2-4, 2003.

[35] A. Chandra, P. Goyal and P. Shenoy. Quantifying the Benefits of Resource Multiplexing in On-Demand Data Centers. 1st Workshop on Algorithms and Architectures for Self-Managing Systems (Self-Manage 2003), San Diego, California, USA. June 11, 2003.

[36] A. Chandra and P. Shenoy. Effectiveness of Dynamic Resource Allocation for Handling Internet Flash Crowds. Technical Report TR03-37, Department of Computer Science, University of Massachusetts, USA. November 2003.

[37] S. Chandra, C. Ellis and A. Vahdat. Differentiated Multimedia Web Services using Quality Aware Transcoding. IEEE INFOCOM 2000, pp. 961-969, Tel-Aviv, Israel. March 26-30, 2000.

[38] X. Chen, H. Chen and P. Mohapatra. ACES: An Efficient Admission Control Scheme for QoS-Aware Web Servers. Computer Communications, Vol. 26 (14), pp. 1581-1593. September 2003.

[39] H. Chen and P. Mohapatra. Overload Control in QoS-aware Web Servers. Computer Networks, Vol. 42 (1), pp. 119-133. May 2003.

[40] L. Cherkasova and P. Phaal. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. IEEE Transactions on Computers, Vol. 51 (6), pp. 669-685. June 2002.

[41] W. Chiu. Design for Scalability. IBM white paper. September 2001. http://www-106.ibm.com/developerworks/websphere/library/techarticles/hvws/scalability.html

[42] J.D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48-59, Welches, Oregon, USA. August 3-4, 1998.

[43] C. Coarfa, P. Druschel, and D. Wallach. Performance Analysis of TLS Web Servers. 9th Network and Distributed System Security Symposium (NDSS'02), San Diego, California, USA. February 6-8, 2002.

[44] J. Corbalan and J. Labarta. Improving Processor Allocation through Run-Time Measured Efficiency. 15th International Parallel and Distributed Processing Symposium (IPDPS'01), pp. 74-80, San Francisco, California, USA. April 23-27, 2001.

[45] J. Corbalan, X. Martorell and J. Labarta. Performance-Driven Processor Allocation. 4th Operating System Design and Implementation (OSDI'00), pp. 59-73, San Diego, California, USA. October 22-25, 2000.

[46] M. Crovella, R. Frangioso and M. Harchol-Balter. Connection Scheduling in Web Servers. 2nd Symposium on Internet Technologies and Systems (USITS'99), Boulder, Colorado, USA. October 11-14, 1999.

[47] T. Dierks and C. Allen. The TLS Protocol, Version 1.0. RFC 2246. January 1999.

[48] R. Doyle, J. Chase, O. Asad, W. Jin and Amin Vahdat. Model-Based Resource Provisioning in a Web Service Utility. 4th Symposium on Internet Technologies and Systems (USITS'03), Seattle, Washington, USA. March 26-28, 2003.

[49] eLiza web site
http://www-1.ibm.com/servers/eserver/introducing/eliza/

[50] S. Elnikety, E. Nahum, J. Tracey and W. Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. 13th International Conference on World Wide Web (WWW'04), pp. 276-286, New York, New York, USA. May 17-22, 2004.

[51] Empirix Solutions for Web Application Performance
http://www.empirix.com

[52] EPCC web site
http://www.epcc.ed.ac.uk/

[53] D. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790, IBM Watson Research Center. October 1994.

[54] A. Ferrari. JPVM: Network Parallel Computing in Java. 1998 ACM Workshop on Java for High-Performance Network Computing, Palo Alto, California, USA. February 28 - March 1, 1998.

[55] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. Hypertext Transfer Protocol -- HTTP/1.1. RFC 2616. June 1999.

[56] A. O. Freier, P. Karlton, and C. Kocher. The SSL Protocol, Version 3.0. November 1996.

[57] D. Garcia, D. Carrera, E. Ayguadé and J. Torres. Eines per a la Monitorització i el Traceig de Servidors d'Aplicacions J2EE. Research Report number: UPC-CEPBA-2004-3, March 2004.

[58] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994.

[59] A. Goldberg, R. Buff and A. Schmitt. Secure Web Server Performance Dramatically Improved by Caching SSL Session Keys. Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), Madison, Wisconsin, USA. June 23, 1998.

[60] W. Grosso. Aspect-Oriented Programming and AspectJ. Dr. Dobbs Journal. August 2002.

[61] J. Guitart, V. Beltran, D. Carrera, J. Torres and E. Ayguadé. Characterizing Secure Dynamic Web Applications Scalability. 19th International Parallel and Distributed Symposium (IPDPS'05), Denver, Colorado, USA. April 4-8, 2005.

[62] J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. Dynamic Resource Provisioning for Self-Managed QoS-Aware Secure e-Commerce Applications in SMP Hosting Platforms. To be submitted to the 20th International Parallel and Distributed Symposium (IPDPS'06), Rhodes Island, Greece. April 26-29, 2006.

[63] J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. Session-Based Adaptive Overload Control for Secure Dynamic Web Applications. 34th International Conference on Supercomputing (ICPP'05), pp. 341-349, Oslo, Norway. June 14-17, 2005.

[64] J. Guitart, D. Carrera, V. Beltran, J. Torres and E. Ayguadé. Session-Based Adaptive Overload Control for Dynamic Web Applications in Secure Environments. Research Report number: UPC-DAC-RR-2005-14, March 2005.

[65] J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta. Tuning Dynamic Web Applications using Fine-Grain Analysis. 13th Euromicro Conference on Parallel,

Distributed and Network-based Processing (PDP'05), pp. 84-91, Lugano, Switzerland. February 9-11, 2005.

[66] J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta. Successful Experiences Tuning Dynamic Web Applications using Fine-Grain Analysis. Research Report number: UPC-DAC-2004-3 / UPC-CEPBA-2004-2, January 2004.

[67] J. Guitart, X. Martorell, J. Torres and E. Ayguadé. Application/Kernel Cooperation Towards the Efficient Execution of Shared-memory Parallel Java Codes. 17th International Parallel and Distributed Symposium (IPDPS'03), Nice, France. April 22-26, 2003.

[68] J. Guitart, X. Martorell, J. Torres and E. Ayguadé. Improving the Performance of Shared-memory Parallel Java Codes Using Application/Kernel Cooperation. Research Report number: UPC-DAC-2003-1 / UPC-CEPBA-2003-1, January 2003.

[69] J. Guitart, X. Martorell, J. Torres and E. Ayguadé. Efficient Execution of Parallel Java Applications. 3rd Annual Workshop on Java for High Performance Computing (part of the 15th ACM International Conference on Supercomputing ICS'01), pp. 31-35, Sorrento, Italy. June 17, 2001.

[70] J. Guitart, X. Martorell, J. Torres and E. Ayguadé. Improving Java Multithreading Facilities: the Java Nanos Environment. Research Report number: UPC-DAC-2001-8 / UPC-CEPBA-2001-8, March 2001.

[71] J. Guitart, J. Torres, E. Ayguadé and J.M. Bull. Performance Analysis Tools for Parallel Java Applications on Shared-memory Systems. 30th International Conference on Supercomputing (ICPP'01), pp. 357-364, Valencia, Spain. September 3-7, 2001.

[72] J. Guitart, J. Torres, E. Ayguadé and J. M. Bull. Performance Analysis of Parallel Java Applications on Shared-memory Systems. Research Report number: UPC-DAC-2001-01 / UPC-CEPBA-2001-1, January 2001.

[73] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Instrumentation Environment for Java Threaded Applications. XI Jornadas de Paralelismo, pp. 89-94. Granada, Spain, September 12-14, 2000.

[74] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. 2nd Annual Workshop on Java for High Performance Computing (part of the 14th ACM International Conference on Supercomputing ICS'00), pp. 15-25, Santa Fe, New Mexico, USA. May 7, 2000.

[75] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Last Results using the Java Instrumentation Suite. Research Report number: UPC-DAC-2000-56 / UPC-CEPBA-2000-25, September 2000.

[76] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Preliminary Experiences using the Java Instrumentation Suite. Research Report number: UPC-DAC-2000-25 / UPC-CEPBA-2000-12, April 2000.

[77] I. Haddad and G. Butler. Experimental Studies of Scalability in Clustered Web System. Workshop on Communication Architecture for Clusters (CAC'04) (in conjunction with International Parallel and Distributed Processing Symposium (IPDPS'04)), Santa Fe, New Mexico, USA. April 26, 2004.

[78] I. Haddad. Scalability Issues and Clustered Web Servers. Technical Report. Concordia University. August 13, 2000.

[79] I. Haddad. Open-Source Web Servers: Performance on Carrier-Class Linux Platform. Linux Journal, Volume 2001, Issue 91, page 1. November 2001.

[80] M. Harchol-Balter, B. Schroeder, N. Bansal and M. Agrawal. Size-based Scheduling to Improve Web Performance. ACM Transactions on Computer Systems (TOCS), Vol. 21 (2), pp. 207-233. May 2003.

[81] HAT: Heap Analysis Tool
https://hat.dev.java.net/

[82] IBM Corporation. AIX V4.3.3 Workload Manager. Technical Reference. February 2000.

[83] IBM Corporation, Microsoft Corporation and VeriSign Inc. Web Services Security (WS-Security) Specification. Version 1.0.05. April 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/

[84] Jakarta Tomcat Servlet Container
http://jakarta.apache.org/tomcat

[85] Java Grande Forum Benchmarks Suite
http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/

[86] M. Ji, E. Felten and K. Li. Performance Measurements for Multithreaded Programs. ACM SIGMETRICS Performance Evaluation Review, Vol. 26 (1), pp. 161-170. June 1998.

[87] G. Judd, M. Clement, Q. Snell and V. Getov. Design Issues for Efficient Implementation of MPI in Java. 1999 ACM Java Grande Conference, pp. 58-65, San Francisco, California, USA. June 12-14, 1999.

[88] M.E. Kambites. Java OpenMP: Demonstration Implementation of a Compiler for a Subset of OpenMP for Java. EPCC Techical Report EPCC-SS99-05, September 1999. http://www.epcc.ed.ac.uk/ssp/1999/ProjectSummary/kambites.html

[89] A. Kamra, V. Misra and E. Nahum. Yaksha: A Controller for Managing the Performance of 3-Tiered Websites. 12th International Workshop on Quality of Service (IWQoS 2004), Montreal, Canada. June 7-9, 2004.

[90] K. Kant, R. Iyer, and P. Mohapatra. Architectural Impact of Secure Socket Layer on Internet Servers. 2000 IEEE International Conference on Computer Design (ICCD'00), pp. 7-14, Austin, Texas, USA. September 17-20, 2000.

[91] I . H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. A. Konstan, D. J. Lilja and P.C. Yew. JaViz: A Client/Server Java Profiling Tool. IBM Systems Journal, Vol. 39 (1), 2000, pp. 96-117.

[92] D. Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[93] R. Klemm. Practical Guideline for Boosting Java Server Performance. 1999 ACM Java Grande Conference, pp. 25-34, San Francisco, California, USA. June 12-14, 1999.

[94] S. Kounev and A. Buchmann. Performance Modeling and Evaluation of Large-Scale J2EE Applications. 29th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG-2003), Dallas, Texas, USA. December 7-12, 2003.

[95] L. Lewis. Managing Business and Service Networks, Kluwer Academic Publishers, 2001.

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

[97] Z. Liu, M. Squillante and J. Wolf. On Maximizing Service-Level-Agreement Profits. 3rd ACM Conference on Electronic Commerce (EC 2001), pp. 213-223, Tampa, Florida, USA. October 14-17, 2001.

[98] M. Malzacher and T. Kochie. Using a Web application server to provide flexible and scalable e-business solutions. IBM white paper. April 2002. http://www-900.ibm.com/cn/software/websphere/products/download/whitepapers/performance_40.pdf

[99] B. Marsh, M. Scott, T. LeBlanc and E. Markatos. First-Class User-Level Threads. 13th ACM Symposium on Operating System Principles (SOSP'91), pp. 110-121, Pacific Grove, California, USA. October 13-16, 1991.

[100] X. Martorell, J. Corbalan, D.S. Nikolopoulos, N. Navarro, E.D. Polychronopoulos, T.S. Papatheodorou and J. Labarta. A Tool to Schedule Parallel Applications on Multiprocessors: the NANOS CPU Manager. 6th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2000) (in conjunction with the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2000)), pp. 55-69, Cancun, Mexico. May 2000.

[101] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé. A Library Implementation of the Nano Threads Programming Model. 2nd EuroPar Conference, pp. 644-649, Lyon, France. August 26-29, 1996.

[102] D. Menasce, V. Almeida, R. Fonseca and M. Mendes. Business-Oriented Resource Management Policies for e-Commerce Servers. Performance Evaluation, Vol. 42 (2-3), pp. 223-239. September 2000.

[103] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, 1994.

[104] Metamata Inc. JavaCC: The Java Parser Generator http://www.metamata.com/JavaCC

[105] N. Meyers. PerfAnal: A Performance Analysis Tool
http://developer.java.sun.com/developer/technicalArticles/Programming/perfanal/

[106] Microsoft Active Server Pages
http://www.asp.net

[107] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. Workshop on Internet Server Performance (WISP'98) (in conjunction with SIGMETRICS'98), pp. 59-67. Madison, Wisconsin, USA. June 23, 1998.

[108] R. Mraz. SecureBlue: An Architecture for a High Volume SSL Internet Server. 17th Annual Computer Security Applications Conference (ACSAC'01), New Orleans, Louisiana, USA. December 10-14, 2001.

[109] MySQL
http://www.mysql.com

[110] Nanos web site
http://www.cepba.upc.es/nanos/

[111] J. Oliver, E. Ayguadé and N. Navarro. Towards an Efficient Exploitation of Loop-level Parallelism in Java. 2000 ACM Java Grande Conference, pp. 9-15, San Francisco, California, USA. June 3-5, 2000.

[112] J. Oliver, E. Ayguadé, N. Navarro, J. Guitart, and J. Torres. Strategies for Efficient Exploitation of Loop-level Parallelism in Java. Concurrency and Computation: Practice and Experience (Java Grande 2000 Special Issue), Vol.13 (8-9), pp. 663-680. ISSN 1532-0634, July 2001.

[113] OpenMP web site
http://www.openmp.org/

[114] OptimizeIt Enterprise Suite
http://www.borland.com/optimizeit/

[115] Pallas GmbH. Vampir - Visualization and Analysis of MPI Resources. 1998.
http://www.pallas.de/e/products/

[116] Paraver
http://www.cepba.upc.es/paraver

[117] W. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides and J. Yang: Visualizing the Execution of Java Programs. International Seminar on Software Visualization 2001, pp. 151-162, Dagstuhl Castle, Germany. May 20-25, 2001

[118] PHP Hypertext Preprocessor
http://www.php.net

[119] E.D. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T. S. Papatheodorou and N. Navarro. Kernel-level Scheduling for the Nano-Threads Programming Model. 12th ACM International Conference on Supercomputing (ICS'98), pp. 337-344, Melbourne, Australia. July 13-17, 1998.

[120] E.D. Polychronopoulos, D.S. Nikolopoulos, T.S. Papatheodorou, X. Martorell, J. Labarta and N. Navarro. An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors. 12th International Conference on Parallel and Distributed Computing Systems (PDCS'99), pp. 148-155, Fort Lauderdale, Florida, USA. August 18-20, 1999.

[121] POSIX Threads. IEEE POSIX 1003.1c Standard, 1995.

[122] P. Pradhan, R. Tewari, S. Sahu, A. Chandra and P. Shenoy. An Observation-based Approach Towards Self-Managing Web Servers. 10th International Workshop on Quality of Service (IWQoS 2002), pp. 13-22, Miami Beach, Florida, USA. May 15-17, 2002.

[123] Quest Software Solutions for Java/J2EE
http://www.quest.com/

[124] S. Ranjan, J. Rolia, H. Fu and E. Knightly. QoS-Driven Server Migration for Internet Data Centers. 10th International Workshop on Quality of Service (IWQoS 2002), pp. 3-12, Miami Beach, Florida, USA. May 15-17, 2002.

[125] E. Rescorla. HTTP over TLS. RFC 2818. May 2000.

[126] A. Serra, N. Navarro and T. Cortés. DITools: Application-level Support for Dynamic Extension and Flexible Composition. USENIX Annual 2000 Technical Conference, pp. 225-238, San Diego, California, USA. June 18-23, 2000.

[127] S. Shende and A. Malony. Performance Tools for Parallel Java Environments. 2nd Annual Workshop on Java for High Performance Computing (part of the 14th ACM International Conference on Supercomputing ICS'00), pp. 3-13, Santa Fe, New Mexico, USA. May 7, 2000.

[128] Silicon Graphics Inc. IRIX Admin: Resource Administration. Document number 007-3700-005, http://techpubs.sgi.com. 2000.

[129] Silicon Graphics Inc. Origin200 and Origin2000 Technical Report. 1996.

[130] I. Subramanian, C. McCarthy and M. Murphy. Meeting Performance Goals with the HP-UX Workload Manager. 1st Workshop on Industrial Experiences with Systems Software (WIESS 2000), pp. 79-80, San Diego, California, USA. October 22, 2000.

[131] Sun Microsystems. Enterprise Java Beans Technology (EJB)
http://java.sun.com/products/ejb

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

[133] Sun Microsystems. Java 2 Platform, Standard Edition (J2SE)
http://java.sun.com/j2se

[134] Sun Microsystems. Java Native Interface (JNI)
http://java.sun.com/products/jdk/1.4.2/docs/guide/jni/index.html

[135] Sun Microsystems. Java Secure Socket Extension (JSSE) http://java.sun.com/products/jsse

[136] Sun Microsystems. Java Servlets Technology http://java.sun.com/products/servlet

[137] Sun Microsystems. JVM Tool Interface (JVMTI) http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html

[138] Sun Microsystems. Solaris Resource Manager[tm] 1.0: Controlling System Resources Effectively. 2000. http://www.sun.com/software/white-papers/wp-srm/

[139] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors, 12th ACM Symposium on Operating System Principles (SOSP'89), pp. 159-166, Litchfield Park, Arizona, USA. December 3-6, 1989.

[140] B. Urgaonkar and P. Shenoy. Cataclysm: Handling Extreme Overloads in Internet Services. Technical Report TR03-40, Department of Computer Science, University of Massachusetts, USA. November 2004.

[141] B. Urgaonkar, P. Shenoy and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. 5th Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, Massachusetts, USA. December 9-11, 2002.

[142] D. Verma. Supporting Service Level Agreements on IP Networks, Macmillan Technical Publishing, 1999.

[143] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. IBM Systems Journal, Vol. 39 (1), 2000, pp. 82-95.

[144] T. Voigt, R. Tewari, D. Freimuth and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. 2001 USENIX Annual Technical Conference, pp. 189-202, Boston, Massachusetts, USA. June 25-30, 2001.

[145] A. Voss. Instrumentation and Measurement of Multithreaded Applications. Thesis. Institut fuer Mathematische Maschinen und Datenverarbeittmg, Universitaet Erlangen-Nuemberg. January 1997.

[146] Websphere web site http://www-3.ibm.com/software/info1/websphere/index.jsp

[147] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. 4th Symposium on Internet Technologies and Systems (USITS'03), Seattle, Washington, USA. March 26-28, 2003.

[148] M. Welsh, D. Culler and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. 18th Symposium on Operating Systems Principles (SOSP'01), pp. 230-243, Banff, Canada. October 21-24, 2001.

[149] Wily Technology Solutions for Enterprise Java Application Management http://www.wilytech.com/solutions/index.html

[150] T. Wilson. E-Biz Bucks Lost under SSL Strain. Internet Week Online. May 20, 1999. http://www.internetwk.com/lead/lead052099.htm

[151] P. Wu and P. Narayan. Multithreaded Performance Analysis with Sun WorkShop Thread Event Analyzer. Authoring and Development Tools, Sunsoft, Technical White Paper. April 1998.

[152] Z. Xu, B. Miller and O. Naim. Dynamic Instrumentation of Threaded Applications. 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), pp. 49-59, Atlanta, Georgia, USA. May 4-6, 1999.

[153] Q. Zhao and J. Stasko. Visualizing the Execution of Threads-based Parallel Programs. Technical Report GIT-GVU-95-01, Georgia Institute of Technology, Atlanta, Georgia, USA. January 1995.