# Complete Instrumentation Requirements for Performance Analysis of Web Based Technologies

David Carrera        Jordi Guitart        Jordi Torres
Eduard Ayguadé        Jesús Labarta
*European Center for Parallelism of Barcelona (CEPBA)*
*Computer Architecture Department, Technical University of Catalonia*
*C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034,*
*Barcelona (Spain)*
*{dcarrera, jguitart, torres, eduard, jesus}@ac.upc.es*

## Abstract

*In this paper we present the eDragon environment, a research platform created to perform complete performance analysis of new Web-based technologies. eDragon enables the understanding of how application servers work in both sequential and parallel platforms offering a new insight in the usage of system resources. The environment is composed of a set of instrumentation modules, a performance analysis and visualization tool and a set of experimental methodologies to perform complete performance analysis of Web-based technologies. This paper describes the design and implementation of this research platform and highlights some of its main functionalities. We will also show how a detailed analytical view can be obtained through the application of a bottom-up strategy, starting with a group of system events and advancing to more complex performance metrics using a continuous derivation process.*

## 1. Introduction

Rapid development of e-business services has extended the use of web and application servers on companies, generating a high demand of tools to design, implement and analyse the applications offering these services. Most of these applications are built around Java (mainly because of its portability and development facilities). Java application servers are becoming a key component in these environments and therefore, the optimization of their behaviour in terms of performance is becoming important. The potential parallelism that exists in concurrent requests as well in the applications offering the service makes the use of parallel platforms necessary. Ensuring an efficient utilization of the resources (e.g. processors, memory, input/output, …) is important in order to achieve good response times and throughput.

Different approaches are used in existing tools to carry on the analysis process of Java applications. All of them report about the behaviour of the applications in terms of object creation, loop structures, execution patterns or time consumption inside functions. However, they do not consider system status nor the interaction between the applications and the underlying operating system. For example, these tools can detect which loop in a program is consuming 80% of the total execution time but they are not able to indicate how much real processor usage has been done inside of that loop because they don't extract low-level information of the system status.

Figure 1 shows the different layers existing in Java-based environments and how their instrumentation is performed by currently existing tools and by our proposal (Java Instrumentation Suite, JIS). Some of the tools oriented to study application servers [6][7][8][9] report different metrics that measure the application server performance, collecting information through the JVM Profiler Interface (JVMPI [1]). This limits the kind of information that they can get and therefore, their ability to perform a fine-grained analysis of the multithreaded execution and the scheduling issues involved in the execution of the threads that come from the Java application. The basic actions performed by current web-based technologies are the following: reading/writing contents from/to disks; receiving/sending data from/to networks; and finally, processing data coming from disks and networks. This implies that, mainly, three basic system resources support web-based technologies and applications: processors, disks and networks. Thus, a general requirement in order to perform complete performance analysis of web-based technologies and applications is to be able to obtain detailed information about the usage of these resources. Currently, the user relies on the use of system utilities (such as *sar*, *mpstat*, or *iostat* on Linux, as proposed in [2]) to collect this information and relate it with the information provided by the above-mentioned tools, thus restricting the analysis possibilities. In addition to that, understanding these

complex systems requires a methodology to derive metrics from the simple resource usage indicators.
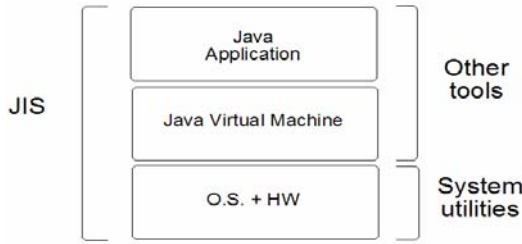


**Figure 1. Layers in Java-based technologies and instrumentation.**

This paper describes the design and implementation of a complete and integrated environment to instrument application servers and analyse and visualize their performance. The eDragon [3] environment is composed of a set of instrumentation tools (JIS), an analysis and visualization tool (Paraver [5]) and an experimental methodology based on a bottom-up strategy over collected data in combination with basic operational analysis concepts (as shown in Figure 2). The environment is being designed to perform an integral analysis of applications based on Java multithreaded technology, specially focusing on application severs and new Web technologies. It can also be used in all Java environments independently of their nature, as it has been successfully tested in Java numerical applications [11][19] as well as on Web servers such as Tomcat [13].
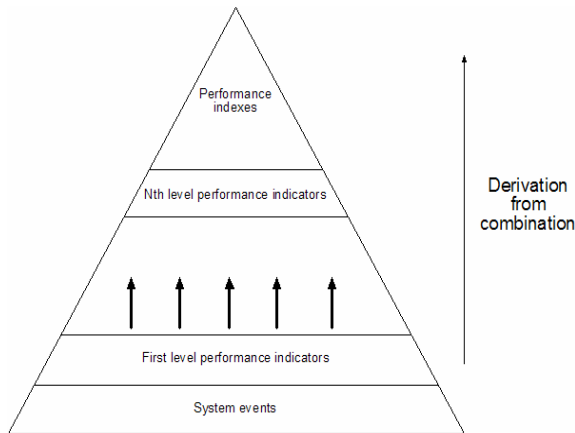


**Figure 2. System performance indexes derivation process**

The remaining of this paper is organized as follows. Section 2 introduces some preliminary concepts about Web-based applications that we consider necessary to follow the paper. Section 3 describes the instrumentation methodology used in eDragon. Section 4 presents the visualization tool (Paraver) and analysis methodology.

Some preliminary experiences using the eDragon environment are described and analysed in Section 5. Section 6 concludes the paper and outlines some future work.

## 2. Basic concepts about web based applications

Applications based on the web, as understood in the scope of this paper, and running on top of application servers are a case of multi-tier application. They are mainly composed of a Client and a Server tier. The client tier is responsible of interacting with application users and to generate requests to be attended by the server; it can be considered as a user interface of the application. The server tier implements the logic of the application and is responsible of serving user-generated requests.

Web contents can be basically divided depending on their nature as static or dynamic. Static contents are those served to clients without any kind of process. HTML files are the best example of static contents: when requested, they are read from disk and sent to clients directly and without any modification. Dynamic contents are those requiring some process before being sent to clients. Typical dynamic contents are server scripts, which are processed and generated results (typically formatted as an HTML page) are sent to clients.

Servlets are more sophisticated dynamic contents than basic server scripts. They are associated with an execution framework composed of auxiliary objects, allowing fast and easy programming of relatively complex algorithms on server machines. Servlets' results can be sent to clients as HTML contents. A typical servlet example is to generate HTML reports based on information extracted from data sources.

Enterprise Java Beans (EJB) are more complex objects than servlets and are the base of porting the object components paradigm to application servers. Currently, J2EE [10] compliant application servers must support EJB objects on their frameworks. As the complexity of web components is increased, also the complexity of implemented applications running on application servers is increased. EJB allow developers to implement real distributed applications based on the Web easily and rapidly. J2EE compliant Application Servers are composed of two different layers: Web container and EJB container, considering servlets as a part of the Web container.

The execution schema of web servers is usually implemented as follows: one thread is responsible of accepting new incoming connections on the server's listening port and assigning to them a socket structure. After this, the acceptor thread assigns the created socket structure to another thread (which will be responsible of

attending and serving received requests) and continues accepting new connections. Threads serving requests are commonly chosen from a pool of threads in order to avoid excessive thread creation overheads. The Tomcat [13] web server used in this paper also follows the described working schema.

## 3. Fine-grain system instrumentation

Instrumentation is the first step when trying to study an already implemented application. Although in some cases source code distributions are available, most commercial applications are not open source. In order to obtain information about the activity of applications, support from the system must be provided. Some environments offer system tools that give information about system status and applications status. Other environments do not offer this kind of facilities and it is required to introduce creative solutions to obtain the desired information.

JIS (Java Instrumentation Suite) is the instrumentation set of tools developed in the eDragon project to study Java-based applications, covering different available platforms [12]. The final result of an instrumentation performed by JIS is an application post-mortem trace, ready to be analysed with Paraver [5]. These traces reflect the activity of each Java thread in the application (through a set of predefined states that are representative of their parallel execution) and collect the occurrence of some predefined events along the whole application lifetime. Paraver traces contain three types of records: state, events and communication. State records represent thread states along time (running, ready or blocked). For example, Figure 3 shows the aspect of a JIS-generated trace visualised with Paraver. On the y-axis application's threads are disposed and on the x-axis time is represented. Event records characterize punctual events in the trace (I/O operations, context switches, socket operations and user-defined events are available on JIS, among others). Finally, communication records are used on JIS to track resource sharing among threads as, for example, sockets being (re)used by different threads.



**Figure 3. Thread state representation on Paraver**

JIS differs from other analysis environments for Java applications in the degree of detail offered about system resources. Most of existing tools [6][7][8][9] focus on offering detailed information about the behaviour of studied applications forgetting the interaction of these applications with the underlying operating system. JIS comes to cover this lack of detail on system status when instrumenting Java applications. The resources observed by JIS are diverse, going from thread status (in relation to CPU state) up to the length and duration of I/O operations.

As said above, the eDragon environment focuses its analysis purposes on application servers and new Web technologies. This fact has forced us to introduce some specific requirements on data acquisition in order to cover all demanded system status aspects necessary to perform a complete modelling process of the behaviour of this kind of applications.

### 3.1. Platforms

Currently, the eDragon environment is available for three platforms: Linux over IA32 architectures, AIX over IBM machines and IRIX running on SGI computers. Particularities of each platform have been overcome on JIS by dividing its architecture in three layers; two of them are system-independent and the other one depends on specific system characteristics, as shown in Figure 4. In this design, platform-independent layers can be reused on different versions of the tool. This paper focuses on the implementation on top of Linux and IA32 architectures, running JDK1.3.
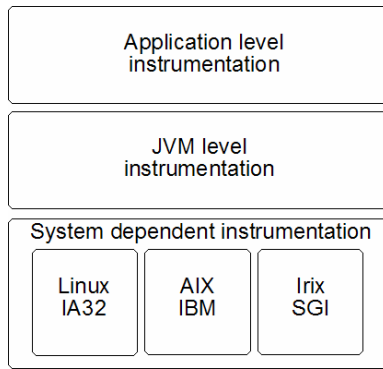
Figure 4. JIS architecture



Figure 5. JIS instrumentation process

## 3.2. Collected information, instrumentation levels and architecture issues

In order to perform a complete system instrumentation allowing a global vision not just of the behaviour of the application but the reasons of that behaviour, it's required to collect information from some different sources. Our proposal focuses on thread status, I/O operations (over storage devices as well as sockets), network devices and memory usage. Information generated by applications through user events must also be considered as an important information source because it allows a put-together process between system status and application behaviour.

Applications can be modelled from their interaction with the system and from their semantics. Interaction between application and system means knowing how application particularities related with the system (number of threads created, degree of activity of these threads, amount of I/O operations done, usage of network resources, …) affect the resulting performance of applications. On another hand, this information should have some degree of relation with application semantics to make it comprehensible. No system activity related with an application can be understood without knowing what was that application doing or trying to do at each moment in time. So, collecting information concerning the behaviour of applications and information about system resources consumption and putting all them together requires a new concept of instrumentation tool, which is one of the contributions of our paper.

All the information sources mentioned above can be classified in different levels depending on how they can be accessed to obtain data. JIS instruments a Java application at three different levels: Application Level, Java Virtual Machine level and System level, as shown in Figure 4. This multi-level architecture is justified by the different nature of data sources implied in the instrumentation process.
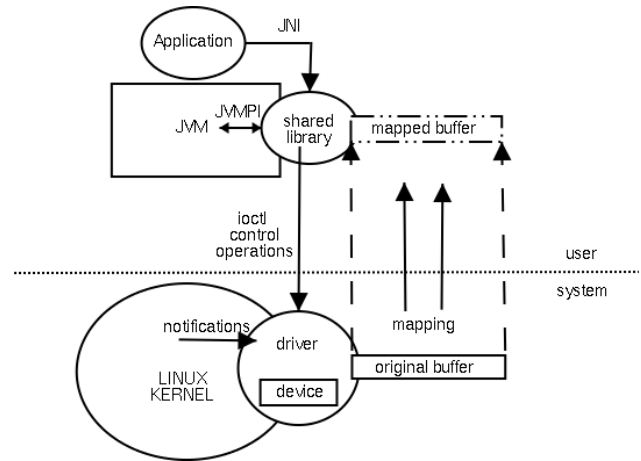
Information collected by the different JIS layers is finally merged in order to produce a final trace. Figure 5 shows how the JIS instrumentation process works. The following subsections describe in detail the architecture of the different JIS levels on the IA32 Linux implementation.

**3.2.1. Application level.** This is the highest level considered by JIS and it is optional (it depends on source code availability). The objective of this level is to generate some especial events when passing by some points on the studied server to introduce some application semantics on the final trace.

It consists in having a shared library, loaded in memory in the same process as the Java Virtual Machine (and, in fact, the Java application), with a common Java Native Interface (JNI [4]). This allows the use of native code (compiled C code on the case of JIS) inside the Java code. In this way, Java written applications can invoke native methods in the shared library that generate JIS events. These events can be visualized inside the trace using Paraver.

Application level instrumentation is specially useful when instrumenting Web servers like Tomcat. User-events are inserted before and after serving requests in order to make this information available to Paraver. This makes possible the identification of individual service requests and to know their time boundaries.

As the source code of applications is not always available, other techniques can be used to extract information about application level on some systems. For instance, when instrumenting IBM WAS (WebSphere Application Server [16]), we use its tracing facility[17]. Other application servers, such as JBoss [18] also implement alternative methods to offer detailed information about their execution that can be used and incorporated to JIS generated traces.

**3.2.2. JVM level.** Java semantics are just considered inside the JVM. Because of this, comprehensive instrumentation of Java applications must include internal JVM information. Current versions of JVM implement a Profiler Interface called JVMPI [1]. JVMPI is a common interface designed to introduce hooks inside JVM code in order to be notified about some predefined Java events. This facility is used by JIS to include information about Java application semantics on its instrumentation process. This means that a developer analysing own applications will be able to see system state information during execution expressed in relation with some of the developed Java application semantics.

The JVMPI is based on the idea of creating a shared library which is loaded on memory together with the JVM and which is 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. Events are notified through a call to a library function that can determine, by parsing received parameters, what JVM event is taking place. The treatment applied to each notified event is decided by the profiler library, but should not introduce too much overhead in order to avoid an excessive slow down of the instrumented applications. Some of available events are: start and end of garbage collecting, class load and unload, method entry and exit and thread start and end.

On JIS, two events are mainly considered to perform application instrumentation. These are Java thread start and Java thread end. Importance of these events comes from their associated information: they contain information about the internal JVM thread name (that one defined by the developer) and allow JIS to match Java threads with kernel threads. Both of them are very useful for developers to understand system information when visualized, because they make it possible to put in relation system extracted data with defined information during development time.

Optionally, other JVM events can be chosen to be incorporated on instrumented information depending on developers' requirements. Activation of many event notifications can result in severe overheads like in the case of the method entry and method exit events, because of their high notification frequency.

**3.2.3. System level.** To perform useful application instrumentation, continuous system state information must be offered to developers. On the Linux version of JIS, considering the open platform characteristics of Linux systems, we decided to extract system information directly from inside kernel. This task was divided in two 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).

The kernel module implements five basic features:

1. Interception of system calls.
2. Implementation of a device driver for the instrumentation device.
3. Creation of an event buffer shareable by system space and user space through a memory map.
4. Creation of a user space system instrumentation control interface through the *ioctl* system call.

Interception of system calls is done by modifying the global system call table in order to use an own function instead of the original system call. After the call is intercepted, the original system call function is invoked in order to preserve the original system behaviour.

The instrumentation driver requires a device that controls it. The driver is implemented inside the Linux kernel module and is used to implement 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. This means that when the JVM notifies to the JIS shared library the start of the shutdown process through the JVMPI, the library indicates to the kernel module that the instrumentation process is concluded, and this communication is done using the *ioctl* call. 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.

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 status: Runnable and Blocked. Runnable implies that a process is ready to run on a processor, but doesn't give information about if it's really running or if it's 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, the kernel patch was used to obtain information about the state of the threads of the system at each moment in time.

A simplified Linux thread state diagram is shown in Figure 6. This is not a complete diagram of possible thread states on Linux, but it is the one considered in JIS. Other states are not considered relevant to study the behaviour of applications in this environment.
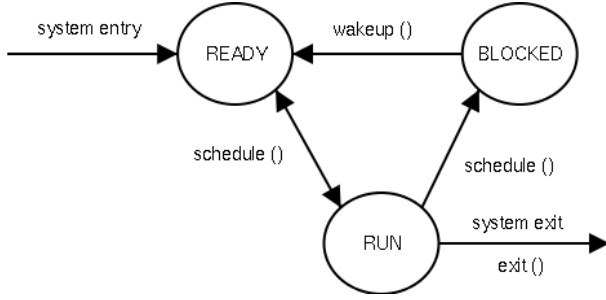
**Figure 6. Thread states considered by JIS and intercepted functions to detect transitions**

**3.2.4. Merging all.** System space and user space captured events must be put together to generate the final trace. The merging process is done when the JVM is shut down. A global memory buffer is allocated, user and system space events are read sequentially, time ordered and inserted to the corresponding buffer position. Finally, the buffer is dumped in order to create the final trace file.

An important issue while merging events is how to share user space and system space buffers in user space. Our decision was to map system space buffer in a user space memory region through the implementation of the *mmap* operation on the instrumentation device. This allows user space processes to work transparently with kernel memory, making it possible to implement a buffer merging process independently of source buffers location.

### 3.3. Overheads

The instrumentation process of JIS introduces some overheads during the execution of the application. Nevertheless, this overhead is low enough not to affect the conclusions extracted from applications analysis. As described in the previous paragraph, once the application execution is finished, the instrumentation library joins the per-thread buffers into a single trace (ordered in time) suitable to be visualized with Paraver. This adds an extra overhead to the whole execution time of the job that does not have any impact in the trace.

In order to measure the overheads of the tool, some experiments have been done. Two kernel applications have been run without instrumentation and with different levels of instrumentation. Their execution times have been studied to determine the impact of the instrumentation on the performance of applications.

The applications that have been used are distinguished by their focus of study: one is CPU intensive and the other is I/O intensive. This first one is a LU benchmark and the second one is the core of the Tomcat Web Server [13] used to transmit disk files (html files) in 2 KB chunks to connected clients. Tests have been repeated with different configurations of the applications, and

obtained results are presented in Table 1 and Table 2. On them, execution times with no instrumentation, with system instrumentation (context switches and I/O operations are captured) and with JVM level instrumentation (obtaining thread names when created) through JVMPI are reported. Times are mean values with corresponding standard deviations. Overheads are indicated between parentheses. As it can be seen, low-order overheads are introduced to execution times when instrumenting applications. Activating JVM information through the JVMPI results in an increase of overheads respect to produced ones with only system level instrumentation. Observed overheads can be considered acceptable in order to not to perturb the behaviour of applications.

**Table 1. CPU intensive application overhead results**

| matrix size | Execution Time (ms) | | System + JVMPI |
| --- | --- | --- | --- |
| | No instrumentation | System instrumentation | instrumentation |
| 1000x1000 | 20662 ± 69,07 | 20710 ± 100,39 (+0,23%) | 20713 ± 23,45 (+0,24%) |
| 750x750 | 9478 ± 32,37 | 9492 ± 20,26 (+0,15%) | 9567 ± 20,81 (+0,93%) |
| 500x500 | 3434 ± 14,82 | 3450 ± 9,79 (+0,47%) | 3506 ± 8,16 (+2,08%) |
| 250x250 | 699 ± 9,02 | 722 ± 13,43 (+3,29%) | 750 ± 1,944 (+8,89%) |

**Table 2. I/O intensive application overhead results**

| Moved size | Execution Time (ms) | | System + JVMPI |
| --- | --- | --- | --- |
| | No instrumentation | System instrumentation | instrumentation |
| 100 Mb | 1918 ± 10,76 | 1995 ± 40,52 (+4,01%) | 2091 ± 48,27 (+9,02%) |
| 200 Mb | 3734 ± 40,26 | 3899 ± 315,67 (+4,43%) | 3957 ± 47,02 (+5,99%) |
| 400 Mb | 14281 ± 201,25 | 14337 ± 165,09 (+0,39%) | 14896 ± 41,67 (+4,31%) |
| 800 Mb | 32093 ± 266,74 | 34931 ± 1547,26 (+8,84%) | 35639 ± 526,87 (+11,05%) |

## 4. Analysis and visualization

Paraver [5] is a flexible performance visualization and analysis tool. 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 behaviour by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver provides a large amount of information useful to improve the decisions on whether and where to invert the programming effort to optimise an application.

Expressive power, flexibility and the capability of efficiently handling large traces are key features addressed in the design of Paraver. The clear and modular structure of Paraver plays a significant role towards achieving these targets.

Paraver offers a minimal set of views on a trace. The philosophy behind the design was that different types of

views should only be supported if they provide qualitatively different types of information. Frequently, visualization tools tend to offer many different views of the parallel program behaviour. Nevertheless, it is often the case that only a few of them are actually used by developers. The other views are too complex, too specific or not adapted to the developer needs.

Following the philosophy of Paraver, derived metrics are simple and powerful: the user can combine two displaying windows of a tracefile using very simple operators (add, product, maximum, etc.) to obtain new semantic functions.

A requirement for Paraver was that the whole operation of the tool has to be very fast in order to make it usable and capable to maintain the developer interest. Handling traces in the range of tenths to hundreds of MB is an important objective of Paraver to enable the analysis of real programs. Easy window dimensioning, forward and backward animation and zooming are supported. Several windows with different scales can be displayed simultaneously. Even on very large traces, the quantitative analysis can be carried out with great precision because the starting and end point of the analysis can be selected on different windows. All these facilities make possible a complete analysis of instrumented applications, qualitatively and quantitatively.

## 5. Experiences with the eDragon environment

The eDragon environment has already been used to perform some exploratory approximations to different consolidated and emerging topics related with Web technologies.

In the following subsections, three different experimental uses of the eDragon are reported to show the analysis capabilities of the proposed environment. First, we use it to visually analyze sources of parallelism in application servers; second, we show how it can be useful to detect situations in which bad resources usage implies degradation in performance; and finally, we show how the detailed information provided by our approach can complement the high-level approaches to system characterization

Before describing them, we present the environment that we have used. As said before, the focus of this proposal is put on application servers. This kind of Java applications are usually accessed through a provided Web interface. Thus, the first experimental scenario for the proposed environment was created through static content services on a Web server. In order to introduce a certain degree of complexity on the studied system, no pure static content Web servers (as apache) were chosen but a Java servlet-based version of Web Servers was considered:

Tomcat 4.0. This Web server presents as a particularity that when running stand-alone, serves static contents through a special dedicated servlet. It makes possible to study the invocation process of servlets on Tomcat through static content services.

In order to perform some realistic experiments with our environment over Tomcat, a static workload generator was required. We use SURGE [14] which generates a static-content workload based on empirical observation of real web sever logs. On another hand, some basic dynamic content tests were done using a servlet-implemented version of a LU reduction kernel.

The instrumentation of Tomcat services was performed using all the different levels of instrumentation that JIS offers. System level instrumentation was turned on while executing the SURGE benchmark, as well as JVM instrumentation used to capture thread names, and application instrumentation through JNI calls to JIS. This last issue was completed with the modification of an extension Java library (implementing functions of the javax package) in order to introduce application-generated events on the final trace indicating the beginning and end of each one of the services made by the Web server.

Tomcat follows a connection service schema where incoming requests are initially attended by an HttpConnector object, which makes an *accept* operation over a socket and chooses an HttpProcessor object from an object pool to process it. Both HttpProcessor and HttpConnector objects contain a background thread inside of them and are the origin of all possible parallelism inside Tomcat.

Some special requirements were imposed when instrumenting applications if Web-based applications were in mind. One of these requirements was to introduce on traces the possibility of following connections since arrived to the server up to being completely served. It was accomplished by identifying connections with one unique identifier assigned sequentially on accepted connections and inserting it on all following socket operations. Additionally, it was necessary to detect activities over sockets from different threads, as when connection acceptor threads assign sockets to request processor threads. It was achieved by inserting Paraver's communication events between threads when sockets "passed" from one thread to others. These events are represented on Paraver as yellow lines going from one thread to other, as visible on Figure 7. All lines are originated on the HttpConnector thread because the original socket accept operations are performed by this thread.

## 5.1 First experimental use

The first experimental use of the eDragon environment was based on doing visual inspections of Paraver traces trying to detect potential sources of parallelism. Traces were generated executing over Tomcat a SURGE-generated workload.

Observing the trace generated when executing the SURGE workload, it's possible to remark some execution properties of this Web server. As shown in Figure 7, the HttpConnector thread (used to accept new incoming connections) is presented on the 13th row. The following threads are the HttpProcessor threads, used to serve received requests. Some HttpProcessor threads (e.g. the first 5 ones in this execution) are created when the server is started. When the number of concurrent requests grows, some new HttpProcessors are dynamically created and added to the thread pool. It shows how Tomcat manages dynamic thread creation in function of system load. Concurrent service of requests is an important source of parallelism exploitation.
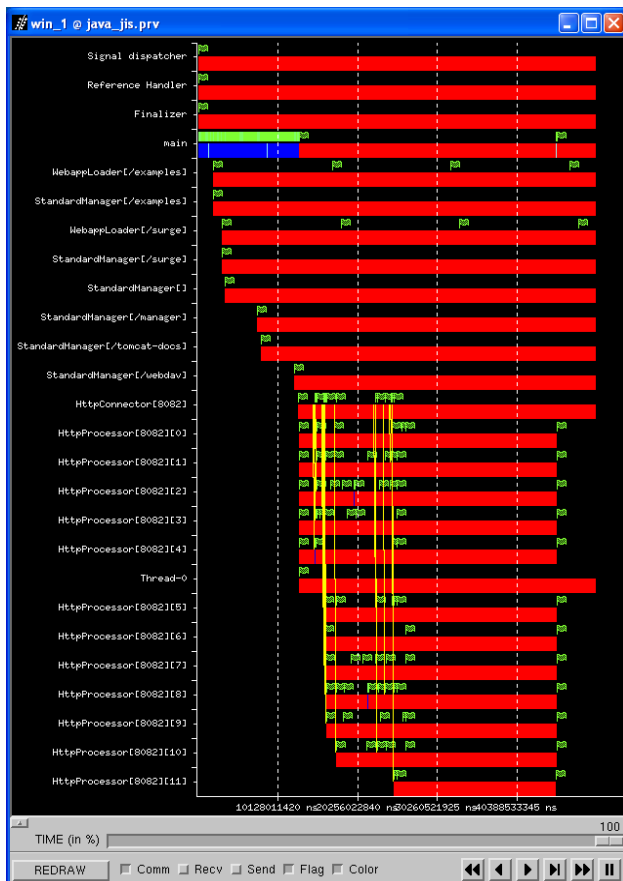


**Figure 7. Paraver visualization of a SURGE generated workload over Tomcat v4.0**

A dynamic web-content approach was also done with the eDragon environment. An LU reduction kernel implemented as a servlet was executed with 2 parallel threads and instrumented. The resulting Paraver trace can be seen in Figure 8. Three threads are really implied in the execution of this kernel. One of them, the 3rd one in the trace, is the application main thread and is responsible of coordinating the work done by the other threads. Working threads can be seen on the 4th and 5th row on the trace and are responsible of performing the LU computation in parallel.
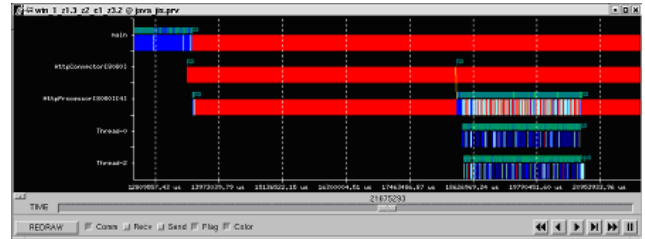


**Figure 8. Paraver visualization of a LUAppl benchmark implemented as a servlet over Tomcat v4.0**

## 5.2 Second experimental use

As a second experimental use of the eDragon environment, it was used to look for performance degradation scenarios caused by bad resource management. A deeper study of some generated traces allowed us to detect some resource problems while serving concurrent requests. For example, Figure 9 shows the sequential service of two concurrent request services. If more resources were available (or better used) the two services could have been attended in parallel. Again, a potential focus of parallelism was detected by a simple visual study of generated traces using Paraver.
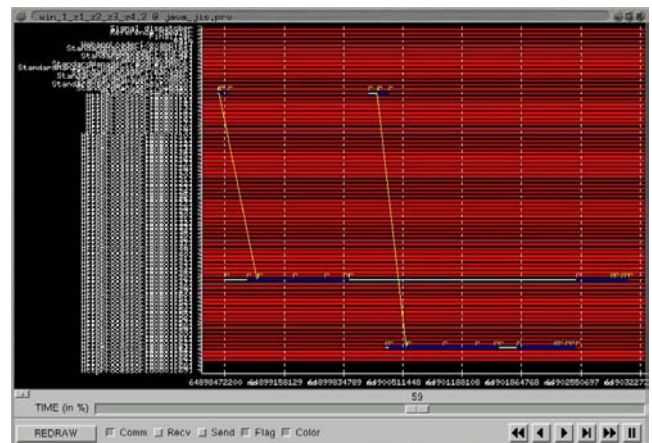


**Figure 9. Concurrent service of requests**

### 5.3 Third experimental use

Finally, a third experimental use of the eDragon environment was done to determine how system load could affect Tomcat performance. First of all, we considered service times by using application-level events inserted before and after a call to the service method of the Servlet class. As all servlets are serviced by invoking this method, it was possible to determine (inserting events on the final trace) where services started and finished. After it was done, we required knowing the size of each one of those services, because we wanted to normalize the service time per connection. Service size was calculated by putting in relation the size of all write operations done over the connection socket during the time of its corresponding service. It was possible by using Paraver statistical facilities. Finally, we determined the system load at each moment by considering it as the number of concurrent connections on service at each moment. As a result, we could consider a size-normalized time per service for each service and put it in relation with the system load at each moment. Resulting numbers are represented in Figure 10. Units are nanoseconds per byte on the y-axis and number of concurrent connections in service on the x-axis. As it can be seen, the number of concurrent connections introduces an important degradation of per-service performance on Tomcat.
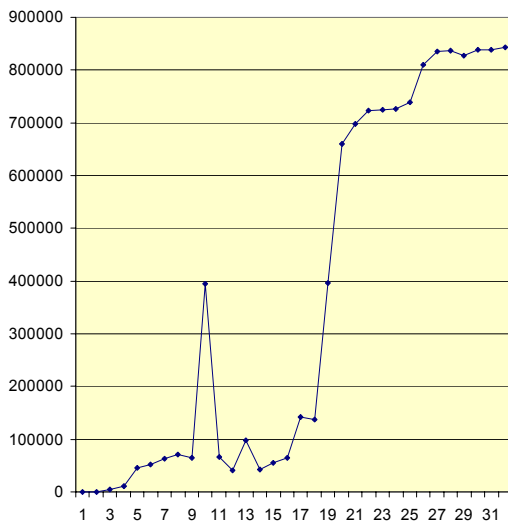


**Figure 10. Response time per byte as a function of system load**

It's important to remark two interesting issues in this plot. First it shows a saturation point of the server when serving 18 concurrent connections. After this point important performance degradation is observed. Second there is a strange behaviour with 10 concurrent connections. It was a good opportunity to check the usefulness of the eDragon environment to determine what happened on the system during that time. It was possible to determine the different periods of time in which ten concurrent connections were being serviced on the system. With this information and doing a qualitative visual analysis of the trace it was possible to detect a long inactivity period on the system, which can be seen on Figure 11. As it can be observed, there's a period of inactivity in all threads of Tomcat. We couldn't determine what was happening on the system during that time, but we can observe that CPU resources were busy working for other system processes. It's important to highlight that this is an important contribution of our environment. Currently available performance analysis tools can't offer this grain of detail of the use of system resources. Most of them could make developers think that there were some programming bugs in their applications, like inefficient loops or bad used locks. With the eDragon environment it's possible to determine that the problem was in the system behaviour and not in the application semantics.
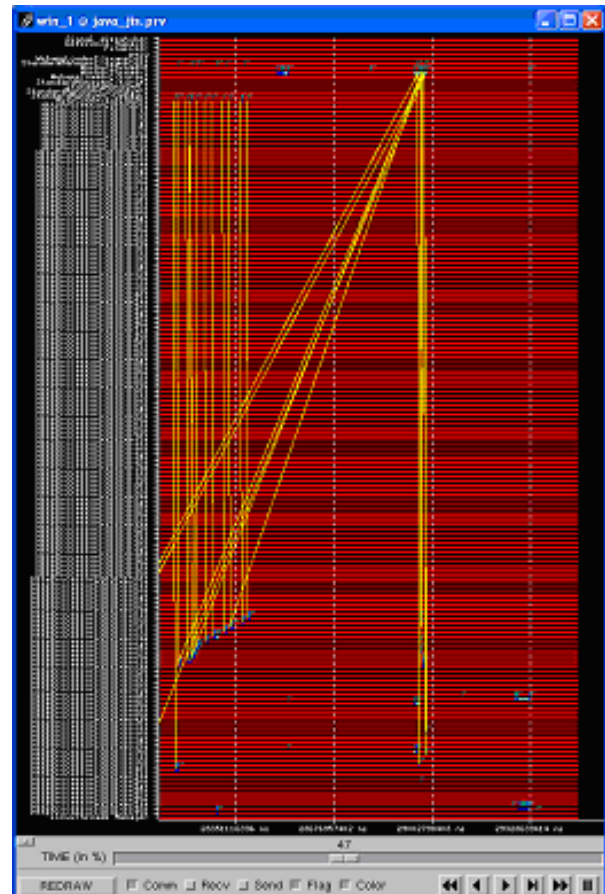


**Figure 11. Paraver visualization of a system inactivity period**

# 6. Concluding remarks and future work

In this paper we have presented eDragon, a research environment that allows a detailed time-analysis of Java applications and a deeper insight into performance analysis. We highlight some conclusions drawn from our initial experimentation that shows the usefulness of the environment.

This instrumentation and analysis is in fact a first step in the eDragon project in the design of a platform for doing research on scheduling mechanisms and policies oriented towards optimising the execution of multithreaded Java Applications Servers on parallel environments focused on new Web paradigms as Web Services and eBusiness extensions.

Our main contribution with this paper is the creation of a new performance analysis environment based on a bottom-up strategy to characterize applications' behaviour. It adds to existing tools the possibility of studying the real state of the underlying system with detail. Our proposal extracts detailed system information in order to perform a complete instrumentation of studied applications. The amount of collected information is variable and can be modified in order to adapt instrumentation to different environments.

The developed environment has been proved in practice with some experimental scenarios and has successfully been used to extract first conclusions about performance on the Tomcat Web server. It has also pointed good perspectives for the optimisation of application servers through coarse-grain parallelization at the level of requests service.

To the best of our knowledge, our system is the first proposal that offers a so fine degree of detail on system resources instrumentation. Benefits of the environment will allow us to entry on new computing fields never explored before with so much detail. From our point of view, an improvement on Application Servers should be reached by introducing a certain grade of cooperation between the operating system and the application server. Following steps should drive application servers to an autonomic optimal working level based on self-tuning techniques.

Future research on application servers' topics will be based in more complex scenarios than the ones presented in this paper (as for example, EJBs and web services). Commercial application servers based on the J2EE technology will be used and specific benchmarks, as the SPECjAppServer [15], will be considered.

# 7. Acknowledgements

# 8. References

[1]     D. Viswanathan and S. Liang, Java Virtual Machine Profile Interface. IBM System Journal, Vol 39, No. 1, 2000.

[2]     E. Cecchet, J. Marguerite and W. Zwaenepoel Performance and scalability of EJB applications ) Proceedings of Oopsla'02 November 4–8, 2002 Seattle, Washington, USA

[3]     eDragon Project. www.cepba.upc.es/eDragon.

[4]     Sun Microsystems. Java Native Interface. March 2000. http://java.sun.com/products/jdk/1.3/docs/guide/jni

[5]     Paraver tool. www.cepba.upc.es/paraver.

[6]     Sitraka JProbe. www.sitraka.com/software/jprobe/

[7]     Performance tuning toolkit for Java-based application development. www.sitraka.com/software/performasure.

[8]     Precise/Insight. A new Perspective on Performance Management. www.precise.com/pdfs/InsightWhitePaper.pdf

[9]     Alexandre Polozoff. Using Introscope for Network Management in Large-Scale WebSphere Application Server Environments. www.wilytech.com/solutions_introscope.html

[10]     Java 2 Enterprise Edition (J2EE). http://java.sun.com/j2ee.

[11]     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 Parallel Processing (ICPP'01), pp. 357-364, Valencia, Spain. September 3-7, 2001

[12]     D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. An Instrumentation Tool for Threaded Java Application Servers. XIII Jornadas de Paralelismo, Lleida, 2002.

[13]     Jakarta Tomcat servlet container. http://jakarta.apache.org/tomcat/

[14]     P. Barford and M. Crovella. "Generating representative workloads for network and server performance evaluation". In Proceedings of ACM SIGMETRICS '98, pages 151--160, Madison, WI, June 1998.

[15]     SPECjAppServer Benchmark. http://www.spec.org/osg/jAppServer.

[16]     WebSphere Application Server. http://www-3.ibm.com/software/webservers/appserv/

[17]     Logs and Traces: WebSphere Application Server. http://www-3.ibm.com/software/webservers/appserv/doc/v40/aee/wasa_content/0803.html

[18]     JBoss EJB server. http://jboss.org.

[19]     J. Oliver, J. Guitart, E. Ayguadé, N. Navarro 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.