An Instrumentation Tool for Threaded Java Application Servers

David Carrera, Jordi Guitart, Jordi Torres, Eduard Ayguadé and Jesús Labarta

Abstract— **Rapid development of e-business services has extended the use of application servers on companies. The Java platform has an important presence on this sector because of its portability and development facilities. Java application servers are becoming a key component in these environments, thus the knowledge of these servers behavior requires the use of new tools to overcome the limitations of existing ones in both offered information and semantics of execution. The natural environment for e-business applications is composed by medium-range parallel servers executing Java based threaded applications. So, understanding threaded Java application servers on parallel environments is the main target of our tool: JIS (Java Instrumentation Suite). This paper describes the design and implementation of JIS and highlights some of the main functionalities. Our initial implementation targets the JVM version 1.3 running Jakarta Tomcat v4.0 on top of a Linux parallel platform with a 2.4.16 kernel.**

Keywords— **Java, Application Server, Instrumentation, Web Services, Parallel Execution**

I. INTRODUCTION

Rapid development of e-business services has extended the use of application servers on companies, generating a high demand on tools to design, implement and analyze applications offering these services. Computing requirements of these kind of applications make necessary the use of powerful machines, basically parallel computers. The maximum performance of this kind of machines is exploited by using a threaded model for applications development.

Although a number of tools have been developed to monitor and analyze the performance of parallel applications [4][10][14][15][19][20], only a few of them target multithreaded Java programs. As studied on [3], different approaches are used to carry on the instrumentation process. Paradyn [20] allows users to insert and remove instrumentation probes during program execution by dynamically relocating the code and adding pre- and post-instrumentation code. Jinsight [14] works with traces generated by an instrumented Java Virtual Machine (JVM). [4] bases their work on the instrumentation of the Java source code, thus requiring the recompilation of the application. TAU [15] acts as JVMPI profile agent [17] with big performance delays.

All of them 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. The Java Instrumentation Suite (JIS) enables this detailed analysis of the application behavior by recording the state of each thread along the execution of the application. The instrumentation is done using system monitoring techniques at the JVM process level (avoiding modifications and recompilation of the source code of neither the JVM nor Java applications).

Originally JIS was a part of Barcelona Java Suite (BJS) [1] at CEPBA [5]. Recently JIS became a part of the eDragon project [7] developed at CIRI [6]. The project is an umbrella project for experimentation and development of technologies for analyzing and optimizing the performance of Dynamic E-Business Applications. The environment integrates techniques and tools proposed in different research frameworks, enabling the exploitation of their combined potential and the development of new proposals.

Current implementations of the JVM allow Java threads to be scheduled by the virtual machine 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 virtual machine. From the point of view of the operating system, there is a single process and a single thread; it is up to the virtual machine to handle all the details of the threading API. In the native threads model, threads are scheduled by the operating system that is hosting the virtual machine. Current implementations of JVM tend to use, for performance reasons, the native threads model. JIS allows the analysis of threaded applications running on these two execution models. However it is designed targeting native threads based executions.

JIS has been successfully tested in both Sun and IBM Linux versions of the JVM. All traces shown in this paper have been captured from the IBM implementation of the JVM. Currently, we are developing JIS versions for other platforms, as IBM and IRIX.

The remaining of this paper is as follows. Section II describes the basic components of JIS. Some preliminary experiments using JIS are reported in Section III and IV. Section V concludes this paper and outlines future research activities in this topic.

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

II. JIS ARCHITECTURE

JIS uses traces in order to analyze the behavior of threaded applications. These traces reflect the activity of each Java thread in the application (through a set of predefined states that are representative of the parallel execution) and collect the occurrence of some predefined events along the whole application lifetime.

Generated traces can be analyzed and visualized with Paraver [12]. Paraver is a tool developed at CEPBA to analyze parallel applications from a qualitative and quantitative point of view.

In order to gather useful information, traces must contain continuous detailed system state and comprehensive semantic application information. Both they must be combined to allow developers understanding what are happening in the system and why it is happening. JIS approaches this idea by combining two execution state data extraction levels. One is based on the Java Virtual Machine Profiler Interface (JVMPI) [17] and the other on the Linux Kernel. Information generated by both them is merged to be complemented and produce final trace comprehensive by developers. Both levels work on the same way: as soon as events are captured, they are inserted on a memory buffer for each thread (independent for each level). When the buffer is full, the runtime system automatically dumps it to disk.

A. *User space instrumentation (JVMPI based)*

Java semantics are just considered inside the JVM. Because of this, comprehensive instrumentation of Java applications must be composed, in part, by internal JVM information. Current versions of JVM implement a Profiler Interface called JVMPI that is a common interface designed to introduce hooks inside JVM code in order to be notified about indicated Java events. This facility is used by JIS to include information about Java application semantics on its instrumentation process. This means that a developer analyzing 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 user 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 slowing down instrumented applications in excess.

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 informations 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.

B. *System space instrumentation*

To perform useful application instrumentation, continuous system state information must be offered to developers. Other versions of JIS, previously designed and targeting other platforms [9] used dynamic code interposition in order to capture references to threads library functions for thread state detection during execution.

In our case, 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, LKM). Figures 1 and 2 show JIS architecture schema and interfaces between JIS levels.

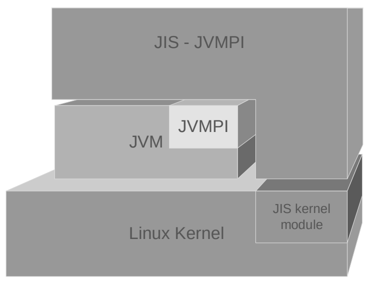

Figure 1. JIS architecture divided on user space and kernel space levels

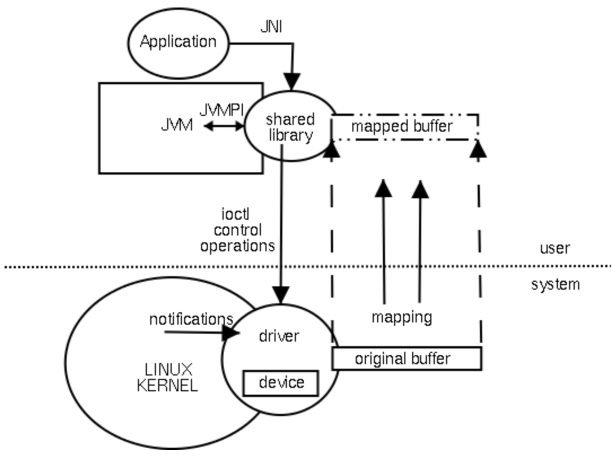

Figure 2. JIS instrumentation process

1) *Kernel module*

The kernel module implements five basic functionalities of JIS:

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

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 behavior.

The instrumentation driver requires a device that controls it. The driver is implemented inside of 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.

Java allows the use of native code (compiled C code on the case of JIS) inside the Java code. This invocation method is called JNI (Java Native Interface) [11]. In order to allow Java developers using JIS to introduce events (similar to checkpoints) inside of their original Java code, a native interface for Java is implemented on JIS. On this way, Java written codes can invoke native methods that finish generating JIS events inside of the user space buffer. These events can be visualized inside the trace using Paraver.

The access to system and user space buffers is done without requiring system locks. Atomic operations are used to fetch and increment pointers indicating the current insertion position on buffers. With this technique, spin locks use inside the kernel is avoided.

2) *Kernel source 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.

Linux offers an interesting way to extract process[2] 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 continuously.

The kernel patch implies that 4 events are captured and inserted in the kernel module allocated buffer from inside the kernel (together with the events generated through system call interception inside the kernel module). Captured events, and their corresponding patched kernel function, are:

1.	Kernel thread creation	(schedule)
2.	Kernel thread destruction	(exit)
3.	Kernel thread block	(schedule)
4.	Kernel thread awake	(wakeup)

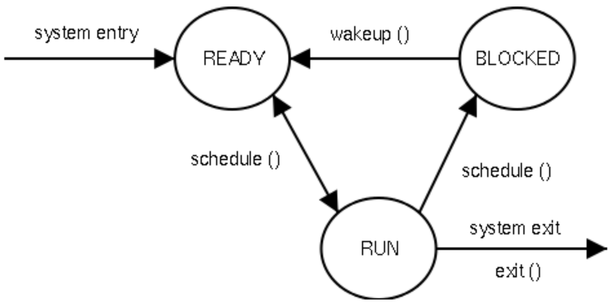

Figure 3. Thread states considered by JIS and intercepted functions to detect transitions

Simplified Linux thread state diagram can be seen on Figure 3. This isn't the complete diagram of possible thread states on Linux, but is the considered one on JIS. Other states are not really relevant to study application behavior. JIS captured thread states are represented on Paraver as shown on Figure 4.

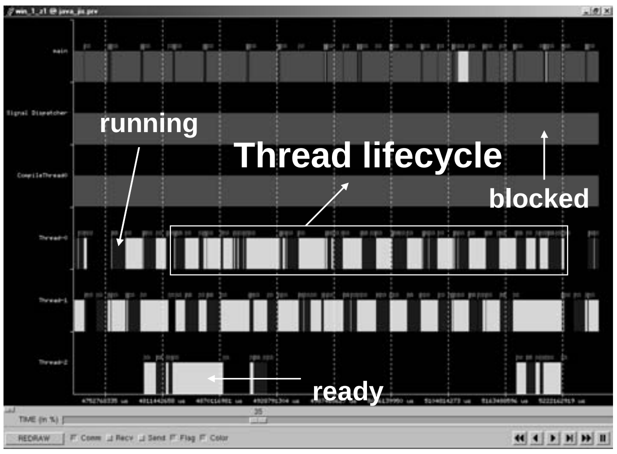

Figure 4. Thread state representation on Paraver

[2] On Linux systems using the linuxthreads implementation of POSIX threads, talking about processes is equivalent to talking about threads because this concrete implementation uses the Linux clone system call to create new threads, which means that threads are, in fact, cloned processes sharing required resources.

C. *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 and user space events and system space events are read sequentially and time-ordered and inserted to the corresponding buffer position.. Finally, the buffer is dumped in order to create the final trace.

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.

III. ANALYZED APPLICATIONS

JIS was designed to allow Java application instrumentation independently of their nature. It performs successful instrumentation of classical numerical applications [8] as well as of conceptually new web applications.

However, the main targeted applications by JIS are related with new web technologies. JIS is oriented towards complete instrumentation of applications constructed over the infrastructure of a web application server. The chosen platform for JIS tests is Tomcat v4.0 [16], because of being the reference implementation of Sun Java Servlets and JSPs specifications and because of its source code availability, that have made possible to insert user events inside of the server to increase comprehension on the behavior of Tomcat.

A. *Web content workload over Tomcat v4.0*

First approximation to application servers is done through static content services. This means serving static HTML pages on Tomcat. This web server presents a special characteristic when running alone (alternatively it can be configured as an extension of Apache Web Server for dynamic content services) and serving static contents: it gives service using a special servlet. This makes possible to study servlet invocation process on Tomcat through static content services.

In order to study Tomcat on a real environment a workload was necessary. The chosen one was SURGE [2]. It generates a workload based on empirical observation of real web server logs.

Some non-static content approximations have also been done to test JIS capacities on typical application server environments. Executing LUAppl benchmark taken from [13] as a servlet over Tomcat has been one of the selected dynamic-content tests. LUAppl consists on a LU reduction kernel over a two-dimensional matrix of double-precision elements

B. *IBM Web Services Toolkit over Tomcat v4.0*

Web services are a still in construction new concept that tries to group a few technologies and put them to work together in order to create a new global platform for e-business. Interoperation, portability and location independency are some of the key ideas behind Web Services.

eDragon project is orienting their research topics towards Application Servers. Web Services, as a particular web application seen from the server side, results an interesting case to study.

Three protocols are the basis of Web Services. These are:

1. SOAP: a lightweight XML based protocol for exchange of information in a decentralized, distributed environment.
2. UDDI: is an XML-based framework that provides standard APIs by which businesses participating in online exchanges can identify who they are and what types of products or services they provide.
3. WSDL: An XML format for describing network services as a set of endpoints operating on messages containing document or procedure info.

Because Web Services concepts and standards are still under construction, an easy to use platform had to be found in order to begin experimentations. IBM Web Services Toolkit [18] is a packed downloadable set of tools prepared to introduce developers on the world of Web Services and allow them to start testing the system quickly. It's distributed supporting two different platforms on server side: IBM WebSphere Application Server (WAS) and Tomcat v4.0. In our case, Tomcat was chosen because of the same reasons detailed in the previous point.

IV. FIRST RESULTS

In this section, we highlight some conclusions drawn from our initial experimentation with JIS. The main idea is to show the usefulness of the tool in both analyzing the behavior of threaded Java application servers and understanding the behavior of the JVM itself. The current implementation targets the Linux IA32 architecture. The used configuration consists in a tetraprocessor machine with Pentium III processors at 800 MHz and the JVM version 1.3 running Jakarta Tomcat v4.0.

JIS offers different views of the application. On one side, JIS allows a global view of the application, which allow us to recognize, for example, the execution paradigm (master/slave, work queuing...), the application concurrency and/or parallelism degree, ... On another side, JIS allows a more detailed analysis (timing) of the behavior and to detect, among other things, load unbalancing at the thread level or critical points that may create some possible bottlenecks.

A. Web content workload over Tomcat v4.0

Visualization of JIS generated traces on Paraver made visible some of the behavior patterns present on this server. Things like dynamic thread creation as a load function and per connection service timers could be modeled with Paraver facilities.

To allow a visual following of received connections some adaptations had to be done on JIS. The main one was the introduction of the called "communication events" on the trace (originally incorporated on Paraver to represent MPI messages) representing different thread interaction with sockets.

Tomcat follows a connection service schema based on the creation of objects called HttpProcessor. These objects have as function to process incoming connections and serve results directly to clients. Incoming requests are initially attended by an HttpConnector object, which makes the *accept* over a socket and chooses an HttpProcessor object to process it. Both HttpProcessor and HttpConnector objects contain a background thread inside of them.

Relation between HttpConnector and HttpProcessor objects is represented by a communication event. It can be observed on Figure 5 as a vertical line. When a communication occurs between two objects (and threads, in fact) a socket is assigned to a new HttpProcessor object. HttpProcessor threads run concurrently and are the base of parallelism on Tomcat. However, observed results show a low degree of parallelism on requests service possibly because of short duration of static requests and particularities of the workloads (specially important are off times: time of inactivity on clients between requests bursts).

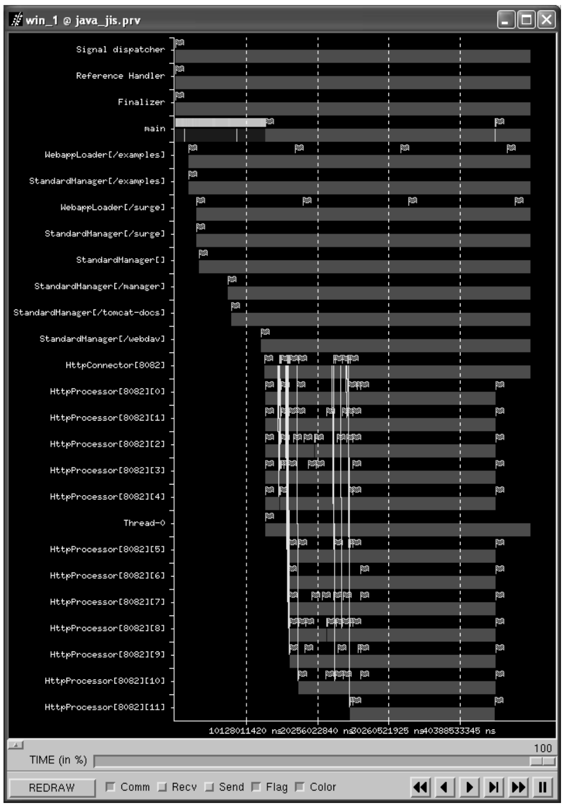

Figure 5. Paraver visualization of a SURGE generated workload over Tomcat v4.0

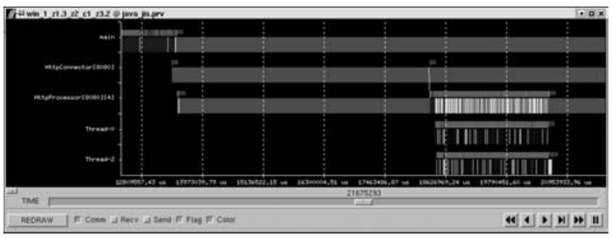

Figure 6. Paraver visualization of a LUAppl benchmark implemented as a servlet over Tomcat v4.0

A non-static content test (a LUAppl benchmark taken from [13] running as a servlet with 2 threads) is shown on Figure 6.

B. IBM Web Services Toolkit over Tomcat v4.0

Web services are composed operations are basically composed by three roles:

1. Service Interface Provider (sip)
2. Service Provider (sp)
3. Service Requestor (sr)

These roles can be identified in Figure 7 on the three marked areas. Information visualized on Paraver is just that one happened on the server. Tomcat was implementing the first two roles and was responding with services execution in the third one (the service requestor). More detailed view of a service requestor role is shown on Figure 8.

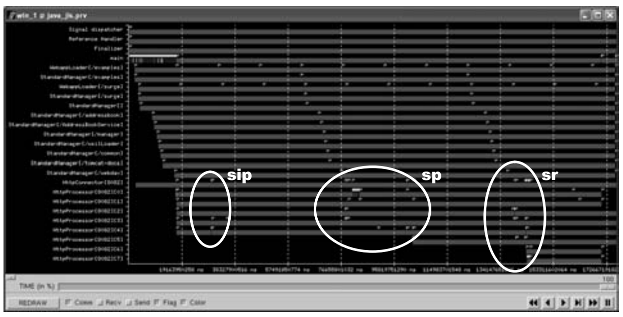

Figure 7. Web Services roles as traced with JIS on the server side

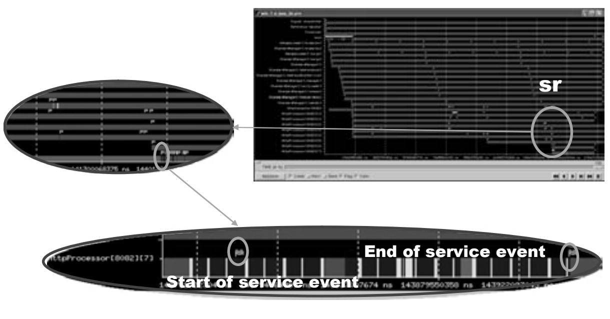

Figure 8. Service Requestor role in detail

C. Overheads study

The instrumentation process of JIS introduces some overhead during the execution of the application.

Nevertheless, this overhead is low enough not to affect the conclusions extracted from applications analysis.

As explained in section II.c, once the application is finished, the instrumentation library joins the per-thread buffers into a single trace (ordered in time) suitable for 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.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have presented the design of JIS (Java Instrumentation Suite), a set of tools designed to analyze the behavior of threaded applications running on the JVM.

JIS allows a detailed time-analysis of the application and visualizes the thread scheduling activity done by the JVM. We have shown the usefulness of the JIS environment with application servers. This instrumentation 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 optimizing the execution of multithreaded Java Application Servers on parallel environments focused on new Web paradigms as Web Services and eBusiness extensions.

Particularities of Application Servers (like sockets reuse by different threads) have been considered to extend traditional instrumentation techniques to allow better comprehension of this kind of applications.

Increasing JIS instrumentation capabilities is an easy task considering its architecture. Adding new system call interceptions should result in more detailed applications description with the advantage of reusing all the structural facilities offered by the tool.

Future work will pass by exploiting JIS to research on new topics. Benefits of the tool will allow us to entry on new computing fields never explored before with so fine grain of detail.

VI. ACKNOWLEDGMENTS

We acknowledge the European Center for Parallelism of Barcelona (CEPBA) and CEPBA-IBM Research Institute (CIRI) for supplying the computing resources for our experiments. This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01 and by Direcció General de Recerca of the Generalitat de Catalunya under grant 2001FI 00694 UPC APTIND.

VII. REFERENCES

[1] Barcelona Java Suite (BJS) http://www.cepba.ucp.es/BJS

[2] 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.

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

[4] A. Bechini and C.A. Prete. Instrumentation of Concurrent Java Applications for Program Behaviour Investigation, In proceedings of 1st Annual Workshop on Java for High-performance Computing, 1999 ACM International Conference on Supercomputing ICS. Rhodes (Greece), June 1999.

[5] European Center for Parallelism of Barcelona (CEPBA) http://www.cepba.upc.es

[6] CEPBA-IBM Research Institute (CIRI) http://www.ciri.upc.es

[7] Barcelona eDragon Project http://www.ciri.upc.es/eDragon

[8] 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

[9] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. 2nd 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.

[10] M. Ji, E, Felton and K. Li. Performance Measurements for Multithreaded Programs. ACM SIGMETRICS/Performance, 1998.

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

[12] J. Labarta, S. Girona, V. Pillet, T. Cortés and L. Gregoris. DiP: A Parallel Program Development Environtment. In proceedings of the 2th. Int. Euro-Par Conference. Lyon (France). August 1996. http://www.cepba.upc.es/paraver

[13] 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. (Also as Research Report number: UPC-DAC-2000-55 / UPC-CEPBA-2000-24, September 2000.

[14] W. Pauw, O. Gruber, E. Jensen, R. Konuru, N. Mitchell, G. Sevitsky, J. Vlissides and J. Yang. Jinsight: Visualizing the execution of Java programs. IBM Research Report, February 2000.

[15] S. Shende and A. Malony. Performance Tools for Parallel Java Environments. Proc.of the Second Annual Workshop on Java for High-Performance Computing, ICS00. St. Fe, New Mexico, May 7, 2000.

[16] Tomcat web site http://jakarta.apache.org/tomcat/

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

[18] IBM Web Services Toolkit http://www.alphaworks.ibm.com/tech/webservicestoolkit

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

[20] Z. Xu, B. Miller and O. Naim. Dynamic Instrumentation of Threaded Applications. In proceedings of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.