Preliminary Experiences using the Java Instrumentation Suite

Jordi Guitart, Jordi Torres, Eduard Ayguadé, José Oliver and 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)

{jguitart, torres, eduard, joseo, jesus}@ac.upc.es

Abstract. The rapid maturing process of the Java technology is encouraging users the development of portable applications using the Java language. As an important part of the definition of the Java language, the use of threads is becoming commonplace when programming this kind of applications. Understanding and tuning threaded applications requires the use of effective tools for detecting possible performance bottlenecks. Most of the available tools summarize the behavior of the application in a global way offering different metrics that are sufficient to optimize the performance of the application in some cases. However, they do not enable a detailed analysis of the behavior of the application; this requires the use of tools that perform an exhaustive and time-aware tracing at a fine-grain level. This paper presents the Java Instrumentation Suite (JIS), a set of tools designed to instrument Java threaded applications using dynamic code interposition mechanism (avoiding the instrumentation and recompilation of the source code and/or the Java Virtual Machine JVM). Our initial implementation targets the JVM version 3.1.1 on top of the SGI Origin2000 parallel platform. The paper describes the design of JIS and highlights some of its main functionalities specifically designed to understand the behavior of Java threaded applications, the JVM itself and to speed them up.

1. Introduction

One of the reasons that support the recent popularity and widespread diffusion of the Java language is that it eases the implementation of parallel and distributed applications. Java offers multithreading facilities through the built-in support for threads in the language definition. The Java library supplies the Thread class definition, and the Java runtime provides support for thread, monitor and condition lock primitives. This convenience and easy-of-use come at the expenses of execution speed of Java programs compared to their equivalent counterparts written in conventional languages such as C++. Recent improvements in all the components that integrate the Java execution platform (bytecode compiler, just-in-time compiler and virtual machine) are effectively reducing the performance gap between applications written in Java and applications written in other languages that benefit from more mature compilation and execution technologies. These improvements and the portability inherent in its definition are leading to an increasing number of threaded applications that are being implemented in Java. However, tuning these applications for performance is mostly responsibility of (experienced) programmers [Klem99].

Although a number of tools have been developed to monitor and analyze the performance of parallel applications [ZaSt95, Voss97, JiFL98, WuNa98, XuMN99, BePr99, Jins00], only few of them target multithreaded Java programs. Different approaches are used to carry on the instrumentation process. Paradyn [XuMN99] allows users to insert and remove instrumentation probes during program execution by dynamically relocating the code and adding pre- and post-instrumentation code. Jinsight [Jins00] works with traces generated by an instrumented Java Virtual Machine (JVM). [BePr99] base their work on the instrumentation of the Java source code, thus requiring the recompilation of the application. 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 dynamic code interposition at the JVM level (avoiding modifications in the source code and its recompilation).

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, the Java threads API is exported to the operating system through a set of layers; threads are scheduled by the operating system that is hosting the virtual machine. JIS allows the analysis of threaded applications running on these two execution models.

To our knowledge, it is the first tool that handles the Java thread scheduling representation of an application. JIS is a part of *Barcelona Java Suite* (BJS) currently under development at CEPBA. The main purpose of BJS is to serve as platform for doing research on scheduling mechanisms and policies oriented towards the efficient execution of multithreaded Java applications on parallel servers.

The remaining of this paper is as follows. Section 2 describes the basic components of JIS. Some preliminary experiments using JIS are reported in Section 3. The paper is concluded in Section 4 providing a list of topics to be considered in the future.

2. JIS basics

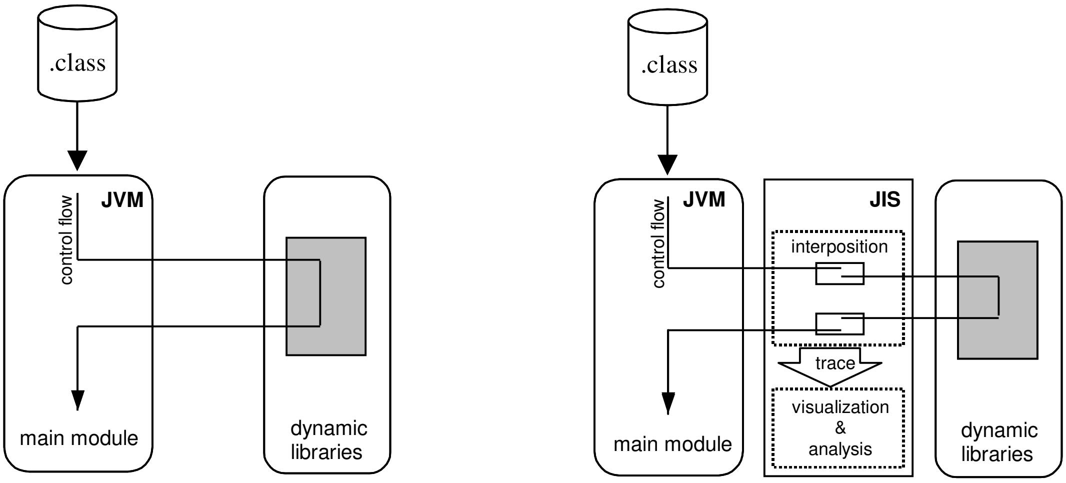
JIS makes use of two components previously developed at the CEPBA: DITools [SeNC00] and Paraver [LGPC96]. DITools is a set of tools bringing an environment in which dynamically linked executables can be extended at runtime with unforeseen functionality. This interposition method is used in JIS to instrument the code of the JVM. References to specific functions of the dynamically linked libraries used by the JVM are captured and probed. This instrumentation does not require any special compiler support and makes unnecessary to rebuild neither the bytecode of application nor the executable of the JVM. The probes executed with the intercepted calls generate a trace file with all the information that is required to analyze and visualize the execution of the threaded application. Paraver serves as the visualization tool. It allows the user to navigate through the trace and to do an extensive quantitative analysis. Figure 1 shows the interaction of the above mentioned components in JIS.

2.1 Application states

We use traces in order to analyze the performance of threaded applications. These traces reflect the activity of each 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. Table 1 summarizes the different states that we consider for a thread.

STATE	**DESCRIPTION**
INIT	Thread is being created and initialized
READY	Thread is ready for running, but there is no processor available for it
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
STOPPED	Thread has finalized

Table 1. Thread states.



(a) (b)

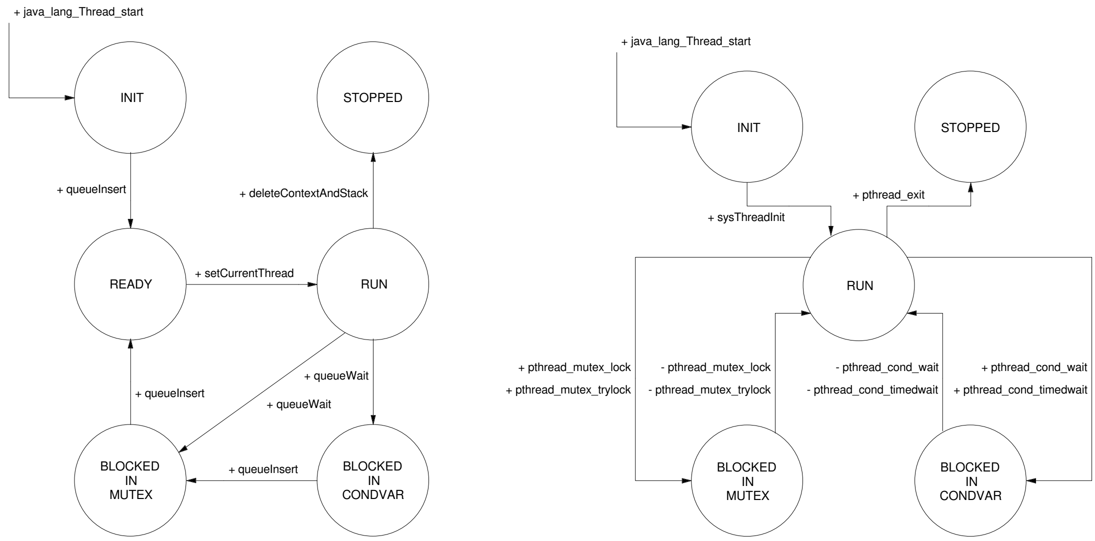
Figure 1. Placement of instrumentation.

2.2 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.

The monitoring 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. 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 2 presents the state transition graph for application threads for both execution models (green and native threads) considered in this work, in which nodes represent states, and edges correspond to procedure calls (indicated by a + sing) or procedure returns (indicated by a - sign) causing a state transition. We can also monitor internal threads of the JVM (like the finalizer, garbage collector,...). In this case, the invocation of other procedures (like `InitializeFinalizerThread`, `InitializeGCThread`, ...) are used to trigger the transition to the `INIT` state instead of the invocation of procedure `java_lang_Thread_start`. This transition graph is then used to derive the interposition routines used to keep track of the state in the performance monitoring backend. These routines are simple wrappers of functions that change the thread state and/or save thread information in the internal structures of JIS. Timestamps associated to transitions and events are obtained from the high-resolution memory-mapped clock offered by the system. The two following codes show simple examples of procedure wrappers:

```
void setCurrentThread_wrapper(args) {
    setCurrentThread_PRE (timestamp, READY, RUN);
    setCurrentThread(args);
    }
int pthread_cond_wait_wrapper(args) {
    pthread_cond_wait_PRE (timestamp, RUN, BLOCKED_IN_CONDVAR);
    ret = pthread_cond_wait(args);
    pthread_cond_wait_POST (timestamp, BLOCKED_IN_CONDVAR, RUN);
    return ret;
    }
```



(a) (b)

Figure 2. State transition graph for (a) green threads and (b) native threads.

3. Preliminary JIS experiences

In this section, we highlight some conclusions drawn from out initial experimentation with JIS. The main idea is to show the usefulness of the tool in both analyzing the behavior of a threaded application and understanding the behavior of the JVM itself. The current implementation targets the SGI Origin 2000 architecture with MIPS R10000 processors at 250 MHz running the JVM version 3.1.1 (Sun Java 1.1.6). Two programs have been used for the purposes of this paper: DHPC_ThreadBench (one of the Java Grande benchmarks from [MaCH99]) and LUAppl (a LU reduction kernel over a matrix of 128x128 double-precision elements taken from [OlAN00]).

JIS offers different views of the same application information. 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.

The DHPC_ThreadBench starts a number of threads that increment a counter and immediately yield. Threads finish their execution when the master thread sets a shared variable (after a specific period of time). The benchmark tests the cases of having 2, 4, 8 and 16 threads. Figure 3 shows one of the windows from Paraver displaying a fragment of the trace from an instrumented run using green threads. Notice that the analysis of the application is done at the thread level: each horizontal line in the window represents the activity of an individual thread, internal to JVM (like CLOCK, IDLE, garbage collector GC, ...) or reflecting the user application itself like (like MAIN, USERn, ...) along the time (horizontal axis, in microseconds). Different colors are used to reflect the different states described in Table 1. Figure 3 shows the point where the MAIN thread is creating 8 additional threads. Notice that two of them (USER7 and USER8) reuse internal thread structures from previous threads and the rest (USER13 to USER18) are new; due to this reuse the JVM takes less time in the creation of the two first ones. We can also observe that at this point, the four threads (USER9 to USER12) that were executing are still queued in the ready queue (they have not finalized their execution because they have not detected that the MAIN thread has updated a global variable that signals their termination). Once the MAIN thread finishes the creation of threads, it awakes these four threads and joins. After that, a sequence in which each of the 8 threads awakes, does some computation and yields follows.

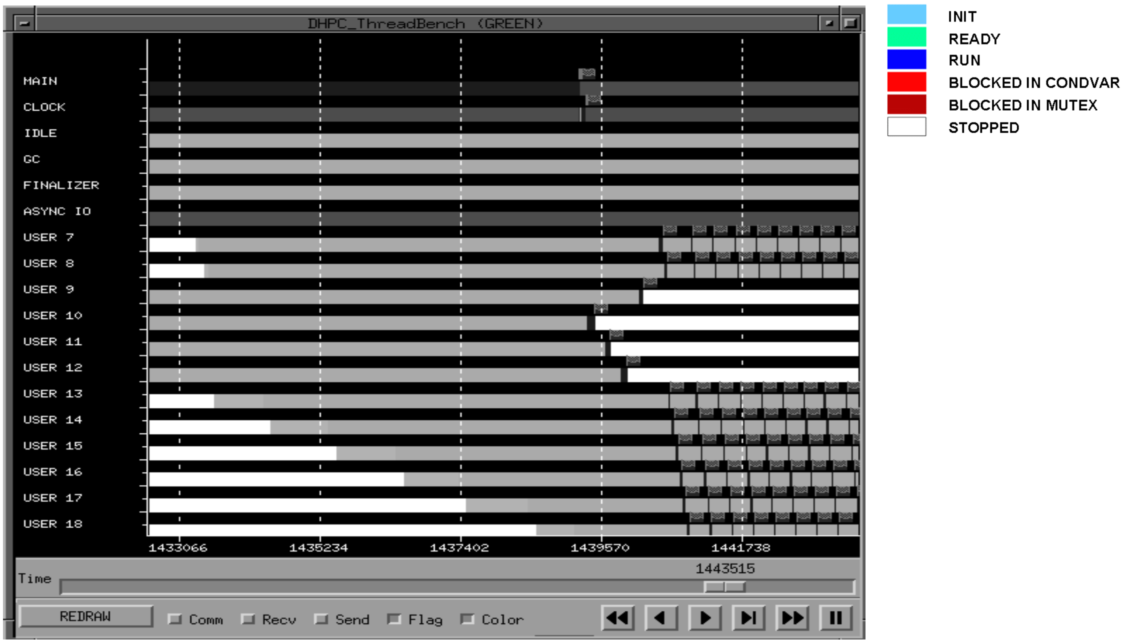


Figure 3: Thread Creation when running the DHPC_ThreadBench with green threads.

Figure 4 shows a Paraver window in which the behavior of the LUAppl application using native threads is shown. Notice that the MAIN thread acts as a master thread that creates four slave threads (USER4 to USER7) along the application lifetime. At this specific point of the trace, each one of the slave threads is executing a subset of the total number of iterations for one of the loops that compose the LU reduction. The analysis at this level reveals the inefficiency, mainly incurred by the JVM itself, in the threaded execution of this part of the application.

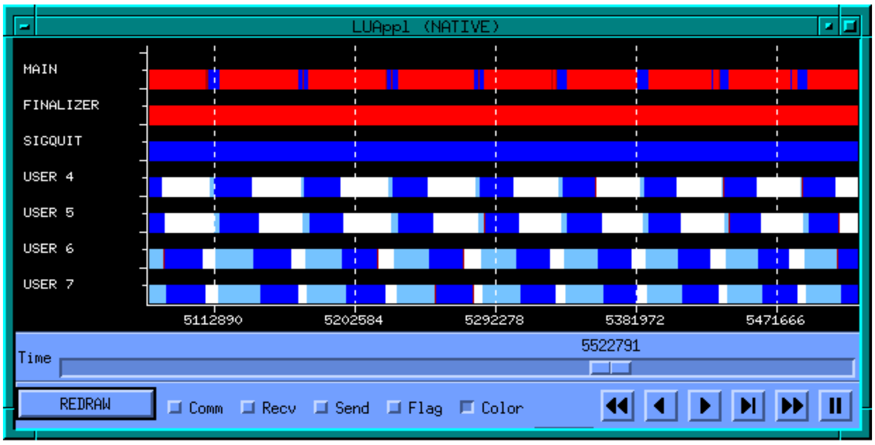


Figure 4: Visualization of the LUAppl running with native threads.

The tracing performed allows the user to interact with Paraver and ask for information related with monitors and critical sections over which specific actions are performed, access to the hardware counters of the system reporting information about the processor and the memory hierarchy behavior, ...

The overhead introduced by the instrumentation process is acceptable. For instance, the instrumentation of LUAppl when executed using green threads incurs an overhead close to 3% in the total execution time. The overhead when using native threads is larger (around 10%) but, from our point of view, acceptable.

4. Conclusions and future work

In this report 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. It uses a tracing mechanism, based on dynamic code interposition that is applied at the JVM level. It does not require access to the source code of the threaded application and does not require any modification in the JVM.

JIS allows a detailed time-analysis of the application and visualizes the thread scheduling activity done by the JVM. This instrumentation is in fact a first step in the design of a platform for doing research on scheduling mechanisms and policies oriented towards optimizing the execution of multithreaded Java applications on parallel servers.

Acknowledgments

We acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying most of the computing resources for our experiments and simulations. Special thanks are to Albert Serra who is the heart in the implementation of the DITools. Ministry of Education of Spain has supported this work under contracts TIC98-511 and TIC97-1445CE.

References

[BePr99] 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.

[JiFL98] M. Ji, E, Felton and K. Li. Performance Measurements for Multithreaded Programs. ACM SIGMETRICS/Performance, 1998.

[Jins00] 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.

[Klem99] R. Klemm. Practical guideline for boosting java server performance. In proceedings of the ACM Java Grande Conference, St. Francisco (CA), June 1999.

[LGPC96] 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.

[MaCH99] J. Mathew, P. Coddington and K. Hawick. Analysis and Development of Java Grande Benchmarks. In Proceedings of the ACM Java Grande Conference, St. Francisco (CA), June 1999.

[OlAN00] J. Oliver, E. Ayguadé and N. Navarro. Towards an efficient exploitation of loop-level parallelism in Java. In Proceedings of the ACM Java Grande Conference, St. Francisco (CA), June 2000.

[SeNC00] A. Serra, N. Navarro and T. Cortés. DITools: Application-level Support for Dynamic Extension and Flexible Composition. In Proceedings of the USENIX Annual Technical Conference, June 2000.

[Voss97] A. Voss. Instrumentation and Measurement of Multithreaded Applications. PhD Thesis. Institut fuer Mathematische Maschinen und Datenverarbeitung, Universitaet Erlangen-Nuernberg, Germany. January 1997.

[WuNa98] P. Wu and P. Narayan. Multithreaded Performance Analysis with Sun WorkShop Thread Event Analyzer. Authoring and Development Tools, Sunsoft, Technical White Paper. April 1998.

[XuMN99] 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.

[ZaSt95] Q. Zhao and J. Stasko. Visualizing the Execution of Threads-based Parallel Programs. Technical Reprot GIT-GVU-95-01, Georgia Institute of Technology, Atlanta, GA. January 1995.