JRastro: A Trace Agent for Debugging Multithreaded and Distributed Java Programs

Gabriela Jacques da Silva¹, Lucas Mello Schnorr¹, Benhur de Oliveira Stein¹

¹ Laboratório de Sistemas de Computação
Universidade Federal de Santa Maria
Faixa de Camobi, Km 9 - CEP 97010-900 - Santa Maria - RS - Brazil
{gabijs,schnorr,benhur@inf.ufsm.br}

Abstract—
Program tracing is one of the most used techniques to debug parallel and distributed programs. In this technique, events are recorded in trace files during the execution of the program for post mortem visualization of its behavior. This article describes JRastro, a trace agent capable of tracing Java programs. The agent was designed to cover three key features: to be transparent to the application developer, to use unmodified Java Virtual Machines and to observe Remote Method Invocations. By integrating these three features, JRastro differentiates itself from similar tools. Unfortunately, for a complete and clean implementation of RMI visualization, additional support on the Java monitoring system is needed.

Keywords— Java program visualization, distributed programs debugging, program tracing, JVMPI

I. INTRODUCTION

Java is a programming language that is being used to develop parallel and distributed applications, probably because it disposes of mechanisms like Remote Method Invocation (RMI) and threads that simplify this programming paradigm. Java is commonly perceived as being execution-inefficient, despite the fact that inefficiency is a property of language implementations, not of the language per se [8].

Despite some mechanisms of this language that facilitate distributed application development, this kind of programming is harder than sequential programming, so it is convenient to use specific debugging tools designed for this kind of programming, for example visualization tools. These tools require the registration of observable information during program execution, called execution traces.

This work describes JRastro and an automatic trace converter. The first is a trace generation agent that allows the visualization of multithreaded and distributed programs developed with Java. The main features of this agent are: transparency to the application developer; no modification of Java Virtual Machine, configurability and to observe Remote Method Invocation. The data offered by the monitoring system used did not allow a complete and clean implementation of the last one (RMI visualization), as will be discussed in section IV.D. The second is an automatic trace converter that was used to facilitate the conversion from JRastro traces to the visualization tool file format. This article is organized as follows: section II talks about parallel program debugging and the visualization tool that is being used in this work. Section III briefly describes some existing tools for tracing Java programs. The fourth section elaborates on the developed trace agent and the fifth section describes the automatic trace converter. The last section presents some conclusions and future works.

II. PARALLEL PROGRAMS DEBUGGING

Parallel program debugging aims at both logic errors correction and performance improvement. Symbolic parallel debuggers extend the functionalities of sequential debuggers to multiple processes. Their main use is on fine-grained debugging. They have difficulties on the presentation of global execution data and the evolution of execution in time. They can also deeply affect a distributed program behavior. For these tasks, a visualization tool is much more suitable, as it shows the processors’ activities and communications. One of the most common graphic representations is made through a time-space diagram, that shows the program behavior during time.

The visualization tool used in this work is Pajé [10]. It fits very well to our problem, because it is independent of any programming model, allowing it to be adapted to the visualization of different programming models in a simple manner. The shape, color and hierarchical relationship of the objects to be visualized are all described in the trace file.

III. MONITORING JAVA PROGRAMS

This section briefly describes JVMPI, the profiling interface available in the Java Virtual Machine, that was used for JRastro development, and also some monitor and visualization tools available for Java programs.

A. JVMPI (Java Virtual Machine Profiler Interface)

JVMPI [11] is an interface that has been incorporated to Java development environments. This interface, supplied by
the virtual machine, allows one to obtain information about
the execution of Java programs. JVMPI has some interesting
properties: it is comprehensive, general-purpose, portable,
has low intrusion and extensible.

JVMPI is considered a comprehensive interface because it
allows tools developed with it to be thread-aware, to verify
monitor contention, to perceive deadlock situations, and
others. The profilers can obtain this information through the
notification of some events by the JVMPI. The utilization of
the JVMPI for information acquisition of Java programs
in execution discharge the virtual machine instrumentation,
since it satisfy the needs of different monitoring techniques,
like trace generation and sampling.

The profiling agent is implemented as a dynamic library.
The virtual machine can be instructed to load it and calls its
functions to communicate the occurrence of events during
the execution of a Java program. The agent receives those
events and can also call functions of the virtual machine, to
obtain more information and to enable or disable the moni-
toring of some kinds of events, according to the agent needs.

B. Monitoring and Visualization Tools

Nowadays, there are several tools for monitoring the ex-
ecution of Java programs and to extract some statistics of
performance and visualization. However, most of them cannot
be used conveniently for performance debugging of dis-
tributed Java applications [6].

A profiling agent for Java programs that comes along JDK
is HPROF. It interacts with the JVMPI and presents profil-
ing information either to the user directly or through profiler
front-ends [5]. This tool is applicable in performance anal-
ysis of sequential applications. However it does not reach
this efficiency on parallel and distributed application analy-
sis, because it does not monitor communications.

JaViz [3] is a profiling and visualization tool that allows
observation of communications, but it uses an instrumented
JVM, forcing the developer to have a modified JVM to de-
bug. TAU [9] is another environment of performance analy-
sis for parallel programs. TAU offers instrumentation tools,
complex parallel programs analysis and measuring. This en-
vironment was extended to enable instrumentation of mul-
tithreaded Java programs and distributed applications based
on mpiJava [1]. However, it does not monitor RMI commu-
nications. Another disadvantage is that it uses the VAMPIR
visualization tool [7], that is proprietary and expensive.

Another visualization tool available to debug distributed
Java programs is VisOK [4]. With VisOK is possible to vi-
ualize the execution of Java programs and its RMI commun-
ications in two ways. The first is a two dimensional ob-
ject interaction diagram and the another is three dimensional
matrix-like object presentation, used for on-the-fly visualiza-
tion. The negative point of this tool is that it modifies the
implementation of RMI of the Java Virtual Machine.

IV. JRAstro Trace Agent

As said in the introduction, the JRastro trace agent was
designed to cover three features:

- transparency on trace generation for the application de-
  veloper;
- no instrumentation on the JVM, avoiding the necessity
  of a modified virtual machine;
- remote method invocation observation, to monitor dis-
  tributed programs that use this technology.

JVMPI was used to permit the Java application developer
to generate traces. This interface was chosen because it helps
to reach the first two features, although currently it does not
provide ways to cover the third one. Thus, the exclusive use
of JVMPI has as a consequence the use of an uninstrumented
virtual machine; that is, user does not need to use a modified
JVM.

In the current state of JRastro, it is possible to trace the
beginning and ending of a thread, local method invocation,
monitor contention, memory utilization and remote method
invocation (with limitations). These events will be described
in details in the following sub-sections, followed by the de-
scription of filtering facilities offered by JRastro.

A. Monitoring of Threads and Local Method Invocation

Multithreading is a commonly used technique in high per-
formance applications, even more in multiprocessed ma-
chines. The visualization of threads should show the time
that each one took to execute some method, the access to
shared objects and the competition to monitors.

The visualization obtained in Paje shows one ruler per
thread (see figure 1). Events generated by a thread are repre-
sented on the thread’s ruler. When a thread starts, an arrow
connects the parent thread to the child thread. When a thread
dies, an arrow connects the ended thread to the one that called
the synchronization routine Thread.join().

Method invocation is also monitored. The visualization
of methods is very useful, because it is a way to verify the
execution time spent on each invocation, as well as verify
what objects are being mostly used. In case of this object
having some synchronized method, it is possible to see if this
synchronization is not being a performance bottleneck.

Figure 1 shows an example of the visualization of trace
data threads obtained with JRastro. The main thread starts
two new threads. These starts two more threads and synchro-
nize them. The main thread is represented by the first ruler,
while the start and the synchronization of the other threads
are represented by arrows. The execution of methods is vi-
ualized by the nested rectangles, different colors represent
different methods. A rectangle embedded in another means
that the method represented by the outer rectangle is calling
the method represented by the embedded one.

B. Monitor Contention

Java uses monitors to allow threads to share data without corrupting them by concurrent access. Monitor visualization permits the developer to see the performance influence of monitor contention and also deadlock situations.

The representation chosen to visualize monitors in Pajé is the following (see figure 2): one ruler represents a monitor, colored when threads are blocked trying to enter it. Arrows connect a blocking thread to the corresponding monitor and the thread that leaves a monitor to the thread that is unblocked. In the moment that a thread tries to access a monitor and it is already being accessed by another thread, the thread and the monitor state are changed to blocked and contended, respectively. The tentative to enter in a monitor is also represented by an arrow that starts on the thread that is trying to do the access and ends on the ruler that represents the monitor. When a thread effectively enters in the monitor, the state of the thread and the monitor are changed back to their previous state, which represents that the thread is no longer blocked and the monitor is being disputed by one thread less. An arrow goes from the thread that was in the monitor to the thread that was blocked trying to access it.

JVMPI notifies three situations related to monitors access and release. In the moment that one thread tries to access a synchronized region that is being used by another thread and is blocked, JVMPI notifies with a JVMPI_EVENT_MONITOR_CONTENDED_ENTER event. This event can indicate bottlenecks in situations that too much threads are blocked in the same monitor. The second situation notified by JVMPI is when a thread effectively enters on the synchronized region after having been blocked waiting for monitor acquisition. The generated event is JVMPI_EVENT_MONITOR_CONTENDED_ENTERED and is always preceded by a JVMPI_EVENT_MONITOR_CONTENDED_ENTER. The association of these two events shows the wait time to enter a monitor. The third event notified is JVMPI_EVENT_MONITOR_CONTENDED_EXIT that is activated when a thread leaves a monitor and another one is waiting to enter the same monitor.

This way of notifying monitor activity pose some problems to obtain the intended representation of figure 2, at least in the virtual machine used. When a thread leaves a synchronized region it usually loses the processor before it registers the monitor release. To observe a program without considering this situation can lead to a wrong

1 The JVM from JDK 1.3.1 from Sun Microsystems.
perception that the synchronized regions are not being executed in mutual exclusion. The adequate solution is to verify the corresponding blocking and blocked threads after the execution of the program. This situation can be better observed on figure 3, where Event A represents the JVMPI_EVENT_MONITOR_CONTENDED_ENTER event, while B is JVMPI_EVENT_MONITOR_CONTENDED_ENTERED and C is the JVMPI_EVENT_MONITOR_CONTENDED_EXIT event.

For example, the B event that informs that thread Z enters the monitor is issued before the C event that says that thread Y left the monitor. That means that from the time of event B to the time of event C, thread Y was not scheduled. The correspondence between threads must be done considering the order of B events, it means, the time that each thread started to execute the synchronized method. This way, it is possible to know that thread Y unblocks thread Z, because it is the last one to execute the synchronized method before thread Z do the same. The identification of the thread blocking Y is only possible when the event C occurs on thread X, because, being the first thread to execute the synchronized method there is no A nor B events registered. Only after the the corresponding threads are found it is possible to generate the arrows indicating the unblocking of one thread by another one.

C. Visualization of Memory Utilization

Monitoring memory allocation and liberation activities can highlight program regions where there is excessive use of memory. This excess can cause a high frequency garbage collector action or too much memory swapping, decreasing application performance.

Figure 4 shows an example of memory allocation visualization in Pajé. Each virtual machine has its own allocation graphic. Memory allocation is shown as a graph growing, while the liberation is represented by a decrease on the line height. Memory liberation can only be observed after the garbage collector action. Garbage collector execution is represented on the ruler that represents the virtual machine execution. When the mouse cursor is over the graph, the amount of allocated memory in that point is shown on the top of the window.

D. Remote Method Invocation

The facility to build distributed programs with Java using the Remote Method Invocation technology (RMI) is one of the main reasons for the growing popularity of this language in distributed applications. The visualization of this kind of invocation can aid the developer to identify bottlenecks, like a server with excessive use.

The intended visualization is shown in figure 5. Rectangles represent the method execution on both client and server threads, in the same way as local method invocation. Arrows are used to connect the two, at the start and end of the method invocation. To achieve this visualization, two things are needed: events that identify the start and end of a remote method invocation on both client and server sides, and some information that allows the matching of these events, because they are independently generated on different virtual machines.

For the events described on previous subsections, most of the needed data is directly available from JVMPI events. RMI communication, on the other side, is ignored by JVMPI. As there is no direct event, it was necessary to identify which methods were executed during a remote invocation. For that, some programs were developed with different utilization schemes of RMI and the information gathered about the execution were analyzed. The calling of a remote method on the client side is identified by the execution of a method whose class name ends in _Stub. On the server side, the identification is done by marking the method as remote in JRastro’s configuration file.

The client and server correspondence was tried by using
only the information available through JVMPI. The basic information for this match is the IP address of the involved machines. On the client side, this information and the server communication port is available. Another data obtained is the object identification that represents the TCP connection with the server. This object is used for future invocations of remote objects on this virtual machine.

On the server side just the IP address of the client is available. Additional data was tried to be obtained by the monitoring of threads created by RMI system and in classes and methods invoked during a RMI call, but nothing else was available. This information could be enough for simple cases, but not for more complex ones. For example, when a host has more than one virtual machine, the server is not able to distinguish between them, as the information available through JVMPI (IP address) is the same for all clients. Besides that, this method would not be portable, because it is based on the implementation of the RMI system (threads created by the system and specific classes and methods) and not on its specification. For all these reasons, this implementation was not proceeded.

In their present state, the trace agent and the trace converter, explained in the section V, allow the visualization of multiple virtual machines in Pájé, but not their interaction. An example of a distributed program visualization is shown in figure 6, where a Java program is composed by two virtual machines, represented by JVM 0 (server) and JVM 1 (client). They interact by RMI calls, that are shown by the arrows. These were added manually to show how a RMI call would be represented.

The RMI representation problem could be solved using instrumentation in other levels, like instrumenting the application code itself, the Java Virtual Machine, or even the operating system. Another solution is an expansion of the JVMPI. The first solution provides a way to obtain all the needed data, but its main drawback is that it obliges the programmer to change his code.

An alternative solution is used by other tracing and visualization tools, like JaViz [3] and VisOk [4] presented on subsection III.B. These tools obtain the necessary information to visualize remote method invocations by using a special, instrumented Java Virtual Machine. The problem of this approach is that it forces the programmer to use a modified virtual machine, that is probably not the one he is using to execute his application. In this way, the programmer may not perceive some implications caused by different implementations of virtual machines.

Another approach is to search for additional data in another level, like the operating system. In this level detailed data about open communication endpoints on the machine in use could be obtained. One possibility is to use the ptrace system call available in UNIX operating systems. This call allows a process to observe the execution of another process, and the system calls that are being invoked, together with its parameters. The problem of both solutions is that an implementation using specific operating systems or virtual machine restricts the portability.

A fourth approach, ideal on the portability issue would be to extend the JVMPI, adding events related to activities that occur on the virtual machine when remote methods are invoked. This activity could be developed with an open source JVM that implements events notification through JVMPI. So, when a remote method call would be executed, this events could be notified with data that allowed remote virtual machine identification. This solution would be clean and portable among virtual machines where these new events were implemented, differently from the previous proposed solutions.

E. Filtering Levels

When using JVMPI, a significant amount of data about internal operations of the JVM can be generated. For example, when configured to notify the invocation of methods, JVMPI notifies the invocation of all methods, including those from the virtual machine’s internal classes. Although the monitoring of certain internal classes may provide useful information for some developers, in most cases it is too much detail. So, it
is necessary to provide some filtering facilities, otherwise the debugging activity would be compromised due to the large amount of information to visualize. In this way, one of the proposals of the JRastro trace agent is to be configurable in various levels. Despite of being possible to filter events to be visualized during trace conversion and on the visualization tool itself, the filtering in trace generation time may reduce trace agent intrusion on the program execution and the size of the trace files.

The first filtering possibility is to choose the classes and methods that will be monitored during program execution. The configuration is done in a file, passed as a parameter when running each virtual machine. With the second filter, the user can choose which events are monitored during an execution. The user can choose, for example, if the memory allocation events and the garbage collection are going to be monitored or not.

Future versions of the tool will include threads and monitors filtering. The necessity of these filters comes from the fact that JVMPI does not differentiate threads created by JVM internal classes from the ones created by user classes. Similar fact happens to monitors, that are also not differentiated. Currently this kind of filtering is done manually during the Pajé visualization. Monitor filtering could be done identifying the synchronized method on the configuration file. The identification of this kind of monitor would be done verifying if the method executed on the moment of a monitor access is synchronized or not.

Figure 7 shows an example of a configuration file. Every line that starts with the letter C represents the class name. The next lines after that, which start with letter M, represent the methods names of the class that will be monitored. The method called init represent the class constructor.

<table>
<thead>
<tr>
<th>C Department</th>
<th>M main R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M print</td>
</tr>
<tr>
<td></td>
<td>M testing</td>
</tr>
<tr>
<td></td>
<td>M &lt;init&gt;</td>
</tr>
<tr>
<td>C Client</td>
<td>M run</td>
</tr>
<tr>
<td></td>
<td>M &lt;init&gt;</td>
</tr>
<tr>
<td>C Employee</td>
<td>M run</td>
</tr>
<tr>
<td></td>
<td>M &lt;init&gt;</td>
</tr>
<tr>
<td></td>
<td>M sync</td>
</tr>
<tr>
<td>C java.lang.Thread</td>
<td>M join</td>
</tr>
<tr>
<td></td>
<td>M start</td>
</tr>
</tbody>
</table>

**V. TRACE CONVERTER**

Visualizing a program execution usually uses a big quantity of information. If all this information is generated during program runtime, program behavior can be greatly disturbed. To reduce this intrusion, tracers try to generate the minimum amount of information that allows the production of the wanted visualization, even if some post-mortem processing is necessary to produce the needed data. The problem of processing this information is generally given to the visualization tool.

This solution, used in many visualization tools, is complex and expensive. First, because the visualization tool must be changed every time the trace file format is changed to accommodate new tracing facilities. Second, because the use of the tool is limited to the visualization of data produced in a specific environment.

A better solution is to have a generic visualization tool with an extensible file format, allowing the visualization of any program that can have its execution trace represented in this format. A generic format is certainly richer than a specific one, because it must contain information describing the data or the way it is to be visualized, together with the data itself. Instead of generating a bigger trace file during program execution, it is better to generate a very specific and small file to avoid intrusion and convert it to a richer representation for visualization later on, to use a generic and portable visualization tool. In this way, only the converter has to be changed/written each time the trace file is changed or every time different information is to be visualized.

Even being smaller and simpler to develop than a visualization tool, a trace file converter is not necessarily an obvious program. We have developed a configurable and extensible trace conversion tool that allows the translation of various trace file formats just by developing a module to read the new trace format and a module to write the output trace format. In this case, a reader to interpret libRastro [2] format and a writer that knows the Pajé format were developed. libRastro was the trace library used to develop JRastro, since it supports multithreaded programs and clock synchronization, essential for distributed programs visualization. As said in section II, Pajé was the visualization tool chosen in this work.

As can be seen in figure 8, the visualization process is characterized by three phases: trace generation, trace conversion and visualization. The first phase, trace generation, is done by JRastro using libRastro. The output of the first phase is a set of trace files that has information delimited in events. Each event has its own type, timestamp and other values according to its type. This events can be extracted of the files by using the trace extraction section of libRastro in the second phase of visualization process. In this phase, the task of the conversion tool is to read all the trace files, to order the events by their timestamps and to translate them to Pajé format. The translation mechanism is a translation machine, controlled by a configuration file. For each event to be translated, this machine reads its type and, according to the rules described in the configuration file, takes actions to translate...
The output of the second phase is a single file that can be visualized by Pajé.

![Fig. 8. Phases and corresponding steps for program visualization](image)

The conversion is based on the event type and on the configuration detailed in the files passed to the converter. For example, during the execution of a Java program, JRastro registers the begin and the end of a method invocation and also the creation of a thread. The trace registered for method invocation is an event with type METHOD_ENTRY, timestamp, identification of the thread that generated the event and the identification of the method invoked. When a method invocation ends, JRastro registers an event with type METHOD_EXIT, timestamp and thread identification. In this case, JRastro does not register the method identification because a METHOD_EXIT always correspond to the last METHOD_ENTRY. The event of creation of a thread has type THREAD_START, timestamp and its identification.

In trace conversion time, the translation machine will read the registered events and then check the configuration provided. In the previous example, the configuration file should have the translation rules shown in figure 9 to translate the events with type METHOD_ENTRY, METHOD_EXIT and THREAD_START.

The translation rules detail how an event must be converted, describing the composition of the new event. As said in subsection IV.A, the representation of a thread in Pajé is a ruler; method invocation is represented by nested rectangles. Method invocation is visualized with two types of events in Pajé, that are PajePushState and PajePopState. The first creates a rectangle and the second finalizes it. The representation of threads can be achieved by Pajé event called PajéCreateContainer, which creates a ruler in the visualization, PajeSetState, to set the first state to the ruler, and PajeEndLink, to finish an arrow. In the example of figure 9, the JRastro event called THREAD_START is converted to these three Pajé events. The field composition in our example is done as follows: the fields Time and Container of all Pajé events, except the PajeEndLink, will have, after the conversion, the value of the correspondent JRastro event field named timestamp and thread_id, respectively. In the case of PajeEndLink the Container will have the value of jvm_id, which is the JVM identification that executes the thread. The field Value of PajePushState will have the value of field method_id of JRastro event. The field Value of PajePushState will have the value of field method_id of JRastro event. The Destination field of PajeEndLink is used to indicate which thread the arrow of visualization will end. To make a complete conversion, it is necessary to write translation rules for all the events that generate useful information for a chosen visualization.

![Fig. 9. JRastro events and the conversion rules passed to the converter](image)

When developer write the translation rules, he does that by composing actions, which are the components of the rules. Each action has its type and other parameters related to this type. For example, in figure 9 the second column shows actions of type event generation. To make the converter more extensible and attractive, other types of actions were built. Such actions are responsible to give to the developer, for example, the power to store information of some JRastro event to use later in the conversion of another event. In figure 9, the field Container of Pajé event PajeEndLink shows this situation, since the value of field jvm_id were registered by another event. Another examples are the “if” action, which executes actions instead of another based on a condition, and actions that operate over stored information.

When the conversion of the events of figure 9 is done along with other converted events, the visualization in Pajé can be similar to the one shown in figure 2. In that figure, the rulers represent threads and the nested rectangles of the rulers show the methods’ executions.

The conversion tool uses configuration files that the trans-
The automatic conversion tool translates the format of the trace file generated during program execution to the format expected by the visualization tool in use. It can also be used to generate alternative visualizations of the same program execution, enhancing the possibilities available to the developer.

With these tools it is possible to visualize in Pajé the creation and finalization of threads, methods invocations, monitor contention, memory allocation and garbage collection. It is also possible to observe the execution of multiple virtual machines, but not all their interaction by remote method invocation.

Planned future development in these tools include the implementation of one of the solutions proposed in section IV.D to solve the problem of lack of information needed to represent remote method invocations, the visualization of synchronization using condition variables and the visualization of other models of communication used in distributed Java programs, like sockets.

VI. CONCLUSION

This work builds an environment that helps developers debug his Java multithreaded and distributed applications. It is composed of a configurable trace agent, called JRastro, and an automatic trace conversion tool. The trace agent was built to cover three key features: transparency on trace generation for the application developer; no instrumentation on the JVM and remote method invocation observation. These features differentiate this tool from similar ones.

The usual visualization with threads (see figure 11) was done with the same set of trace files to compare the visualization by threads and by objects. Alternative visualizations like these are easy to implement just by changing configuration files.

REFERENCES


