A Survey on Java Profiling Tools

Javier Bartolomé, Jordi Guitart

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)

{jbartolo, jguitart}@ac.upc.es

Abstract. In this paper we will present JVMPI, an API created by Sun to facilitate the creation of Java profilers. We will introduce the main characteristics of JVMPI comparing it to the other techniques. We will study the functionality of a representative number of Java profilers: JVMPI-based and non-JVMPI-based. Among the great number of profilers using JVMPI and other techniques, we will present a representative group for each flavor. We have taken Hprof + PerfAnal + HAT, OptimizeIt, jProf and TAU as the representative group for the JVMPI compliant profilers. On the other hand, the representative group of non-JVMPI compliant profilers includes JaViz, Jinsight and JIS.

1. Introduction

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, most of them 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 describes a representative group of these tools, comparing their analysis facilities when improving the performance of Java applications.

This paper is part of the project developed by Javier Bartolomé Rodriguez, which has the main aim to construct a profiler using JVMPI trying to get as much information as the non-JVMPI profilers can get, and using JIS as the model to follow. JIS is a part of Barcelona Java Suite (BJS) [1] 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 introduces the concept and the characteristics of a profiler. Section 3 describes the basics of JVMPI. Some representative JVMPI compliant profilers (Hprof [7] + PerfAnal [10] + HAT [3], OptimizeIt [9], jProf [6] and TAU [12]) are introduced in Section 4, and some representative non-JVMPI compliant profilers (JaViz [4], Jinsight [5] and JIS [2]) are described in Section 5. The paper is concluded in Section 6.

2. Profilers

To analyze the performance of an application we need the help of profilers. Profilers are those tools that show the behavior of an application while it is being executed. The information that profilers show can help us to find the reason of a loss of performance because of a bottleneck.

The profiler or the group of tools that acts as a profiler can include the following basic in order to classify the tools:

- **Instrumentation**: Code may be instrumented automatically (e.g., object code patching or by compilers) or manually by inserting calls to instrumentation library routines. During subsequent execution, the instrumentation library records pertinent performance data, including process, thread, network, I/O, schedule, …
- **Performance Data Extraction**: After instrumentation, performance data is captured from one or more executions. All this information is stored in a trace, file that illustrates the behavior of the program. This file consists of a set of records containing the events that happened during the execution of the program and when these events occurred.
- **Visualization**: After data extraction, performance data is visualized, it could be used another tool that is able to visualize this type of trace files. Such tools can display, after the execution of the program, the trace file highlighting different types of information.
- **Analysis:** During visualization data is analyzed to identify system and application performance bottlenecks. This step requires the developer to analyze the information showed in the previous step and to detect the potential problems. And finally, try to understand why they are occurring.

Once all the information is analyzed, the user should modify the source code trying to fix the possible performance bottleneck. Then the application must be rerun and profiled again to detect new problems.

Profilers in the literature normally performs more than one of the above steps, and can be compound of different tools doing each one some of the steps mentioned before.

3. JVM Profiler Interface

Until the appearance of Java2 Platform to perform profiling of Java applications you should have to work with an instrumented JVM. This JVM normally was given with the profiler that you wanted to use. The instrumented JVM could waste time collecting profiling information that you will not use it later, or collect not enough information of a subject in which you were interested of. These facts provoke that the analysis were mastered by the information that the profiler gives and the way the JVM was instrumented. Moreover if we want to use more than one profiler we will have a copy of a virtual machine for each one.

Since the version 1.2 of JDK Sun presents the Java Virtual Machine Profiler Interface (JVMPI). JVMPI is an interface that profilers can use to obtain profiling information generated from de JVM. This means that all standard JVM is really an instrumented JVM that generates profiling information that can be captured using the JVMPI.

The principal aim in the designing of JVMPI was to be general & powerful enough to suit the needs of different profilers, this can be confirmed if we take a look at the features that present:

- Supports different types of profiling: code instrumentation, statistical sampling
- Produce thread-aware cpu time profiles
- The possibility to enable or disable certain types of events
- A great variety of events that can be collected from the JVM: threads, monitors, objects, classes, …

Figure 1 shows the architecture of a profiler using JVMPI:

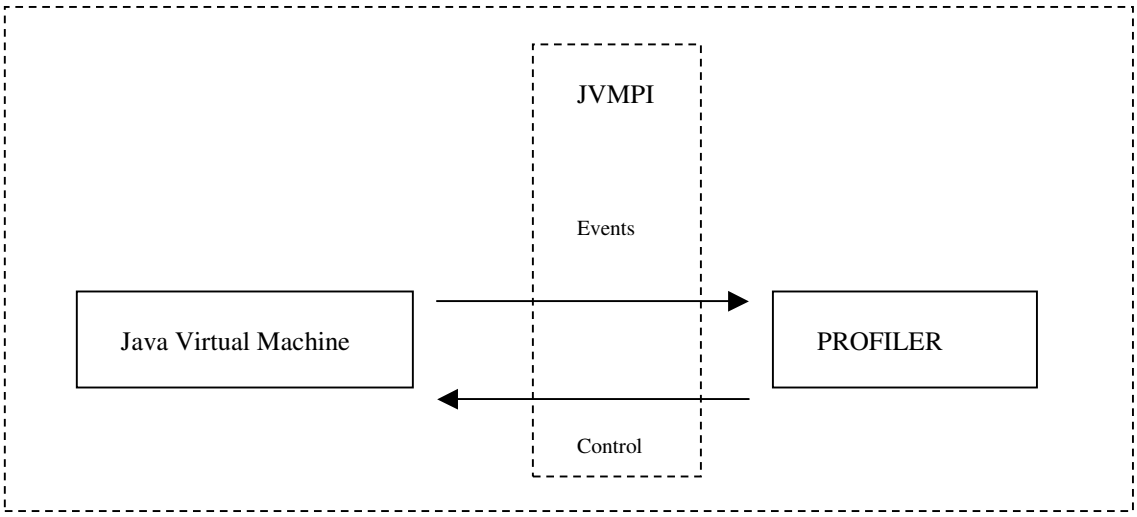

Figure 1. JVMPI Profiler

The profiler is implemented as a dynamically loaded library, when the JVM starts looks for a pre-specified entry point in the profiler and since then, all events generated by the JVM are sent to the profiler in the way that JVMPI describes.

4. JVMPI compliant tools

4.1. Hprof & PerfAnal & HAT

Hprof is a profiler developed by Sun and shipped with Java 2 SDK (Standard Development Kit). Its main objective is to illustrate the power of JVMPI and the way that may be utilized. It's available for each platform that it's available the SDK and the source is available too. The steps covered by this profiler are:

- **Instrumentation**: There's no need to implement an instrumentation because, as we said above, the standard JVM is really an instrumented JVM. Almost all JVMPI profilers rely on the instrumentation produced by the JVM, and then they can focus their effort in the treatment of this information. However, there are some JVMPI-profilers that use an extra instrumentation to get more information, as we will see it later.

```
  Hprof usage: -Xrunhprof[:help]|[<option>=<value>, …]

  Option Name and Value   Description               Default
---------------------     -----------               -------
  heap=dump|sites|all     heap   profiling          all
  cpu=samples|times|old   CPU usage                 off
  monitor=y|n             monitor contention        n
  format=a|b              ascii or binary output    a
  file=<file>             write data to file        java.hprof(.txt for ascii)
  net=<host>:<port>       send data over a socket   write to file
  depth=<size>            stack trace depth         4
  cutoff=<value>          output cutoff point       0.0001
  lineno=y|n              line number in traces?    Y
  thread=y|n              thread in traces?         N
  doe=y|n                 dump on exit?             Y

  Example: java -Xrunhprof:cpu=samples,file=log.txt,depth=3 FooClass
```

Figure 2. Hprof Profiling Options

- **Performance data extraction**: This is the main task of Hprof, it receives the information generated from the JVM and stores it in his internal structures. When the execution has finished, it dumps the information into disk as a text or binary file. Figure 2 shows the options to generate different profiles in Hprof.
- **Visualization**: To visualize the information generated by Hprof we have different options:
  - ✓ Inspect the result file generated by Hprof
  - ✓ Use *PerfAnal* to see the results
  - ✓ Post-process the file generate by Hprof with *Heap Analysis Tool (HAT).*

```
JAVA PROFILE 1.0.1, created Wed Feb 28 02:49:35 2001
THREAD START (obj=80e9fc8, id = 1, name="Finalizer", group="system")
THREAD START (obj=80eab78, id = 2, name="Reference Handler", group="system")
THREAD START (obj=80eb6d0, id = 3, name="main", group="main")
THREAD START (obj=80f04d8, id = 4, name="Signal Dispatcher", group="system")
THREAD START (obj=80f2530, id = 5, name="CompileThread0", group="system")
THREAD START (obj=81293a0, id = 6, name="Thread-0", group="main")
THREAD START (obj=812c9b8, id = 7, name="Thread-1", group="main")
THREAD START (obj=812e938, id = 8, name="Thread-2", group="main")
THREAD END (id = 7)
THREAD END (id = 8)
THREAD END (id = 6)
THREAD END (id = 3)
THREAD START (obj=8128490, id = 9, name="Thread-3", group="main")
THREAD END (id = 4)
TRACE 1: (thread=3)
    <empty>
TRACE 4: (thread=8)
    Amigo.run(thread1.java:Unknown line)
    java.lang.Thread.run(Thread.java:Unknown line)
TRACE 6: (thread=3)
    java.lang.Object.wait(Object.java:Unknown line)
    java.lang.Thread.join(Thread.java:Unknown line)
    java.lang.Thread.join(Thread.java:Unknown line)
    thread1.main(thread1.java:Unknown line)
TRACE 2: (thread=3)
    thread1.main(thread1.java:12)
TRACE 5: (thread=6)
    Amigo.run(thread1.java:Unknown line)
    java.lang.Thread.run(Thread.java:Unknown line)
TRACE 3: (thread=7)
    Amigo.run(thread1.java:Unknown line)
    java.lang.Thread.run(Thread.java:Unknown line)
MONITOR DUMP BEGIN
    MONITOR java.lang.ref.ReferenceQueue$Lock(80ec180) unowned
    waiting to be notified: thread 1
    MONITOR java.lang.ref.Reference$Lock(80ed348) unowned
    waiting to be notified: thread 2
    RAW MONITOR "_hprof_dump_lock"(0x8095d78)
    owner: thread 9, entry count: 0
MONITOR DUMP END
MONITOR TIME BEGIN (total = 0 ms) Wed Feb 28 02:49:39 2001
MONITOR TIME END
CPU TIME (ms) BEGIN (total = 4) Wed Feb 28 02:49:39 2001
rank   self  accum   count trace method
   1 25,00% 25,00%       1     4 Amigo.run
   2 25,00% 50,00%       1     6 java.lang.Object.wait
   3 25,00% 75,00%       1     5 Amigo.run
   4 25,00% 100,00%      1     3 Amigo.run
CPU TIME (ms) END
```

Figure 3. Cpu Profiling with Hprof

Using the first option we will have to use a text editor to see the result of Hprof and understand the meaning of each data expressed. Figure 3 shows the result file generated by Hprof with an example of cpu profiling.

At the end we can see the cpu profile and discover where the time is spent. Taking a look at the trace column we could inspect the part of code that corresponds to each row expressed in the cpu profile.

The second option is using PerfAnal, a graphic tool that takes the output generated by Hprof and present it in a more understandable manner. It interprets the profile data to report the total cost of methods and it presents its results in a graphical format to assist the analysis of the user. Figure 4 shows the typical screenshot of PerfAnal with its classic 4-divided window analysis.

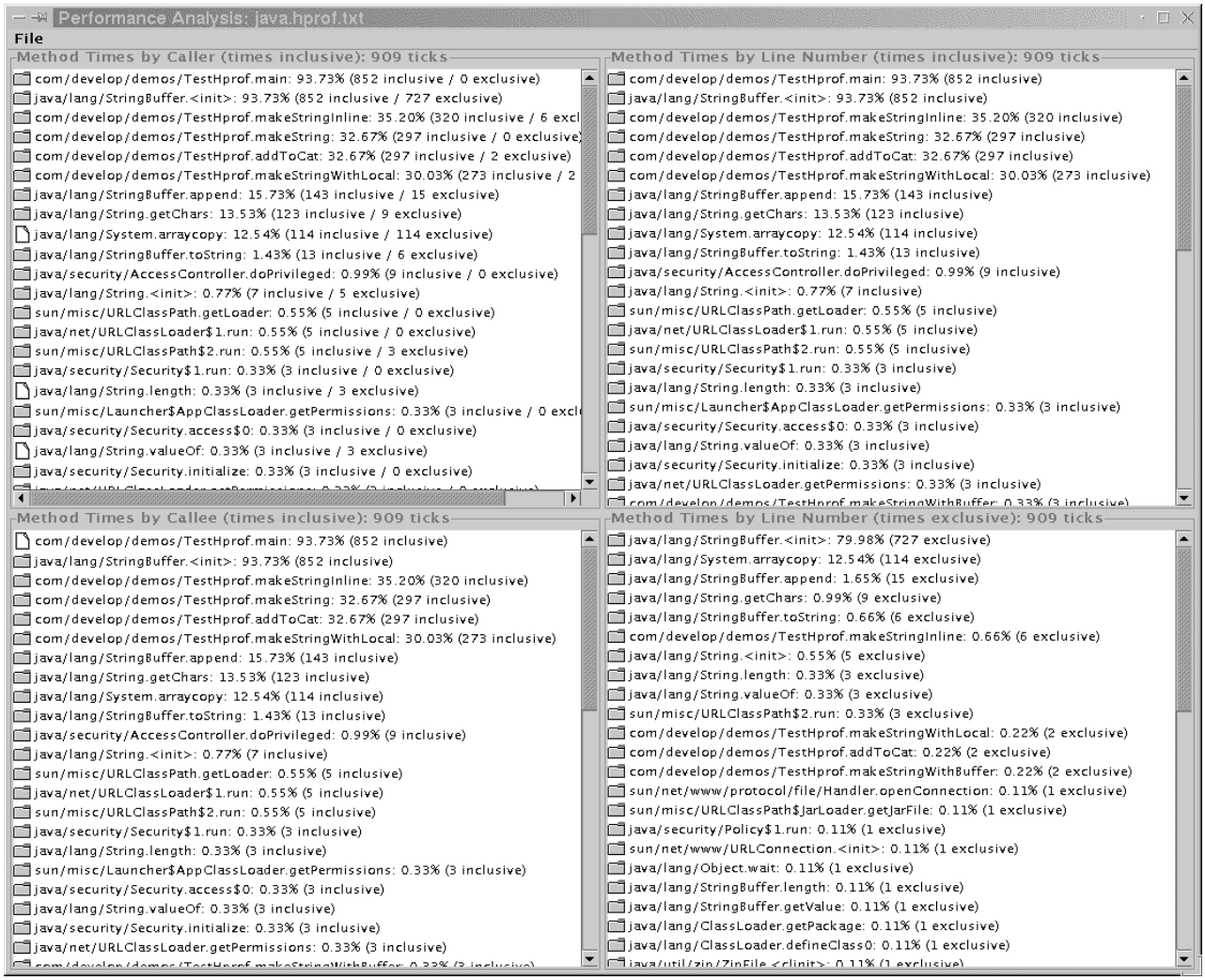

Figure 4. PerfAnal view

Each window shows the same information seen from different perspectives:

- *CPU usage by caller*: It's the upper-left quadrant of the screenshot and presents the time spent in various methods (inclusive of subroutine calls). You can open up branches to explore how the time of a function is divided between the different functions that it calls.
- *CPU usage by callee*: It's the lower-left quadrant and shows the same information that the quadrant explained before, but organizes the analysis by callee.
- *CPU usage by line number*: It's the upper-right quadrant, and presents the same information, but expressing the time spent in each line of the source code.

- *CPU usage, exclusive of subroutines*: It's the lower-right quadrant, it explains really where the time is spent.

Furthermore we can position the cursor in all quadrants on a method that we want to analyze or select a thread that we have interest on study its behavior. Finally, you can save your analysis to disk.

The third option presented above was HAT, a tool also developed by Sun that takes the output generated by Hprof and presents information of how the heap has been used. Before we can use HAT, we have to produce an Hprof output profile in binary mode. HAT starts an HTTP server that analyzes the binary profile and displays the results in HTML format. Figure 5 shows the installation of this server.

```
$ hat hola1.hprof
Started HCODEP server on port 7000
Reading from /home/Javier/hola1.hprof…
Dump file created Tue Mar 05 13:12:27 2001
Snapshot read, resolving…
Resolving 12483 objects…
Chasing references,
    Expect 23 dots ….................
Eliminating duplicate
    references ….............
Snapshot resolved.
Server is ready.
```

Figure 5. HAT server installation

From this point we can consult the information in the port 7000 of our localhost with our browser. (http://localhost:7000).

First of all, the tool presents a list of all the classes and we can select the *Show all members of the rootset* button to discover possible memory leaks. We can also select the *Show instance counts for all classes* button to list the number of instances for each class that was loaded during the execution.

```
5109 instances of class java.lang.String
5095 instances of class [C
2210 instances of class java.util.Hashtable$Entry
968 instances of class java.lang.Class
407 instances of class [Ljava.lang.String;
323 instances of class java.util.SimpleTimeZone
305 instances of class sun.java2d.loops.GraphicsPrimitiveProxy
304 instances of class java.util.HashMap$Entry
269 instances of class [I
182 instances of class [Ljava.util.Hashtable$Entry;
170 instances of class java.util.Hashtable
138 instances of class java.util.jar.Attributes$Name
131 instances of class java.util.HashMap
131 instances of class [Ljava.util.HashMap$Entry;
130 instances of class [Ljava.lang.Object;
105 instances of class java.util.jar.Attributes
```

Figure 6. HAT option '*Show instance counts for all classes*'

In this second view, we can follow the link of a class and HAT will present a line for each instance of that class (a line for each object) and the space in bytes that occupies in memory. Moreover, if we click on one of these instances we can inspect the exact point in which this instance was created.

```
Instances of java.util.SimpleTimeZone

  class java.util.SimpleTimeZone

  java.util.SimpleTimeZone@0x004f48c0 (68 bytes)
  java.util.SimpleTimeZone@0x003d5ad8 (68 bytes)
  java.util.SimpleTimeZone@0x004fae88 (68 bytes)
  …..
  Total of 323 instances occupying 21964 bytes.
```

```
Object allocated from:

  java.util.TimeZoneData.<clinit>(()V) :
        TimeZone.java line 1222
  java.util.TimeZone.getTimeZone((Ljava/lang/String;)
    Ljava/util/TimeZone;) :
    TimeZone.java line (compiled method)
  java.util.TimeZone.getDefault(
        ()Ljava/util/TimeZone;) :
    TimeZone.java line (compiled method)
  java.text.SimpleDateFormat.initialize(
        (Ljava/util/Locale;)V) :
    SimpleDateFormat.java line (compiled method)
```

Figure 7. HAT instance information

Strong points

- Available for every Java2 Platform
- Hprof comes with all the SDK (Standard Development Kit)
- The source code is available and can help developers to construct their own JVMPI-profiler or modify Hprof to obtain some specific information

Weak points

- Execution time when it's profiling
- Difficult to understand the Hprof trace file directly.
- Need to use another tools like PerfAnal or HAT to work with a friendly environment.

More information about this profiler can be found in [7], [10] and [3].

4.2. OptimizeIt

OptimizeIt is a product from Intuitive Systems with full support for Sun's JDK 1.1.x and versions 1.2 and 1.3 without modifications and it allows local and remote profiling of Java programs on multiple platforms. It's available in all flavors of Windows, Solaris and Linux.

To know better this tool we will see the steps that cover this tool:

- **Instrumentation**: The main work of the instrumentation is assigned to JVMPI, but as OptimizeIt can be used with JDK 1.1 (JVMPI were not yet implemented for JDK 1.1), some extra instrumentation is needed. OptimizeIt offers an API to control profiling directly from source code. Moreover the information of profiling is reduced when it's used JDK 1.1.x.
- **Performance data extraction**: OptimizeIt collects all the information generated by JVMPI and stores it in his internal structures. With all this information stored OptimizeIt can show them interactively as they are occurring or store it in a file to do the analysis when the program has finished.

  We will introduce the different types of profiling offered by OptimizeIt:

  - ✓ *CPU profiling*: Provides the two types of cpu profiling that offers JVMPI, sample-based profiler and a instrumentation-based profiler. The first one is used with large

applications that needs minimum overhead. On the other hand, the second one is used with applications that need a great precision arriving to the microsecond precision.

- ✓ *Memory debugging*: To spot the memory leaks collects the object references and highlights the references that need to be cleared, so that can be garbage collected.
- ✓ *Object allocations*: Monitors the object allocations in real-time so that users can detect performance issues such as excessive allocations. Users can track the temporary allocations and which code is responsible for excessive temporary allocations. It is stored instance count and size for each class.

- **Visualization**: The same program contains the graphic visualizer of all information that is collected. Exists a global performance window that displays in real-time high-level performance related data that can help to understand where is the problem. The visualizer has a lot of different forms to display the information. We will show only those that we find the most remarkable. Figure 8 shows a cpu profiling it expresses where the application is spending its time. We can see that the execution of the application is expressed as a tree call, the below window show the possible hot spots, a very useful feature.

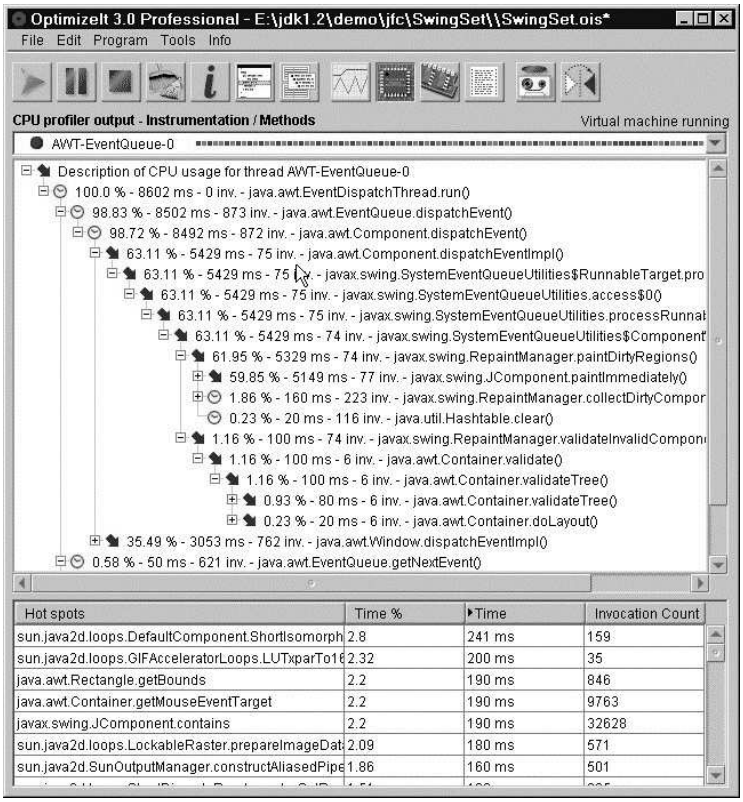

Figure 8. OptimizeIt cpu profiling

In Figure 9 we can see a memory profile, in which classes are ordered by the number of instances that are used in the application. It also shows the space that these instances occupy in memory.

The graphic results of the profiling can be exported in an ASCII or HTML mode. In other remarkable features we can integrate OptimizeIt with application servers like IBM WebSphere and ATG Dynamo.

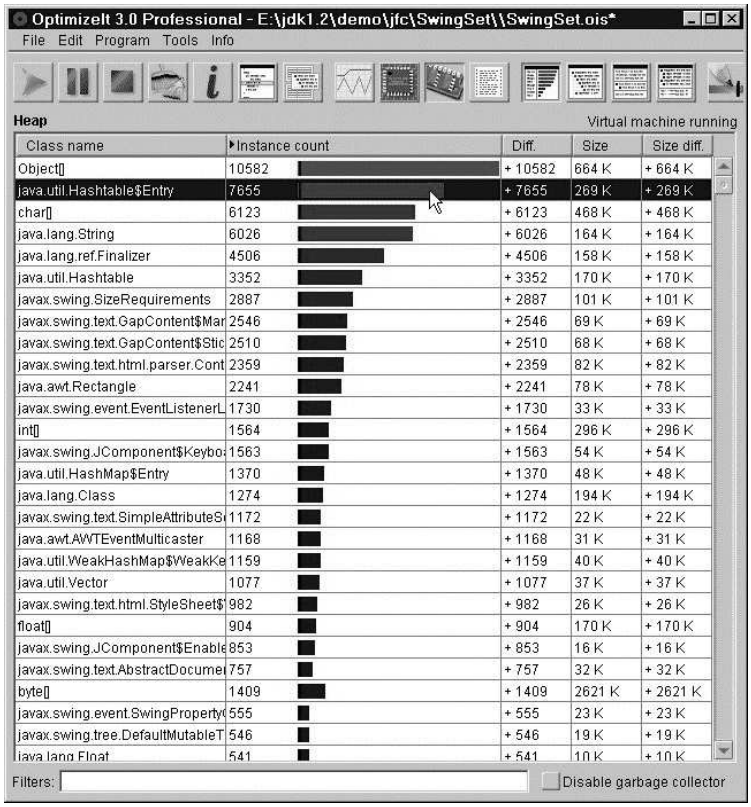

Figure 9. OptimizeIt memory profiling

Strong points

- Compatible with JDK1.1.X and Java2 Platform
- Very good visualizer
- Integration with application servers

Weak points

- Execution time when it's profiling

More information about this profiler can be found in [9].

4.3. jProf

jProf is a non-commercial profiler developed using JVMPI, it was constructed to identify the typical problems that appear in java application developing:

- Excessive memory usage
- Excessive synchronization
- Excessive processor usage

We can download the source of this profiler, although the source code distribution doesn't include all the required files to build a fully working version. Despite of this restriction, you can build the profiler and make corrections to the source to build a variation of jProf. It's available for Windows NT and Solaris platforms.

We will see now the different steps that cover this profiler:

- **Instrumentation**: All the work of the instrumentation is assigned to JVMPI, so there's no need to extend our explanation in this point.

- **Performance data extraction**: The profiler gets the information generated by JVMPI and apart from saved it in its internal structures, it generates at the end an XML profiling result. Figure 10 shows the different types of information that can be collected.

```
D:\jProf\lib> java -XrunjProf:help

Jprof usage: -XrunjProf[:help][<option>=<value>, … ]

Option Name and Default             Description
----------------------              -----------
help=                               Print this help message.
File=jProf.xml                      Write data to file.
Mem_alloc=n                         Profile memory allocations. [y|n]
mem_free=n                          Profile memory release. [y|n]
thr_use=y                           Profile thread use and conflict. [y|n]
mthd_use=n                          Profile method use. [y|n]
core_classes=n                      Profile core classes. [y|n]
mthd_sample=n                       Sample Method profiling.[sample rate|n]
```

Figure 10. jProf options

- **Visualization**: jProf contain a GUI to present the results of a profiling in HTML format from the XML file. You can also run a profiling from the GUI environment, specifying as well the options of the profiling. Later we can inspect the results of the profiling. Figure 11 shows a garbage collector profiling result presented in tables.

Figure 11. jProf table visualization mode

Figure 12 presents the same data in graphic mode. This view can be used to obtain a first impression of where is the problem and then inspect the HTML profiling information to obtain more specific information.

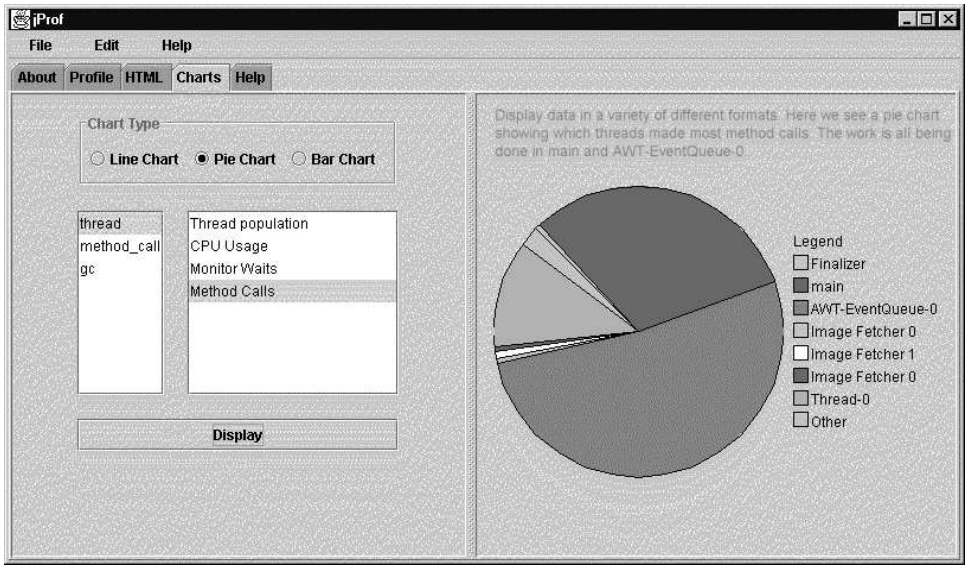

Figure 12. jProf graphic visualization mode

Strong points

- The source code is available, and can be modified to obtain a more specific profiler

Weak points

- Not available for a great number of platforms
- Poor visualizer compared with the commercial tools

More information about this profiler can be found in [6].

4.4. TAU

Tool developed at the Computer & Information Science – University of Oregon. It has been constructed based on JVMPI and rely on other existing tools that allow to analyze the information generated by TAU.

- **Instrumentation**: All the work of the instrumentation is assigned to JVMPI, so there's no need to extend our explanation in this point.
- **Performance data extraction**: The information generated is similar to other tools although is the first tool that generates the profiling information in a format to be used by other visualization tools, for instance RACY or VAMPIR.
- **Visualization**: As said before, the visualization phase is done by other tools, we will show some screenshots because we can get an idea of the information that can be collected. Figure 13 left shows a RACY diagram with the threads in a Java execution with a color-coded to show the method that is in execution in each moment. Figure 13 right shows the relation of the methods and their color with the % of cpu used for each one.

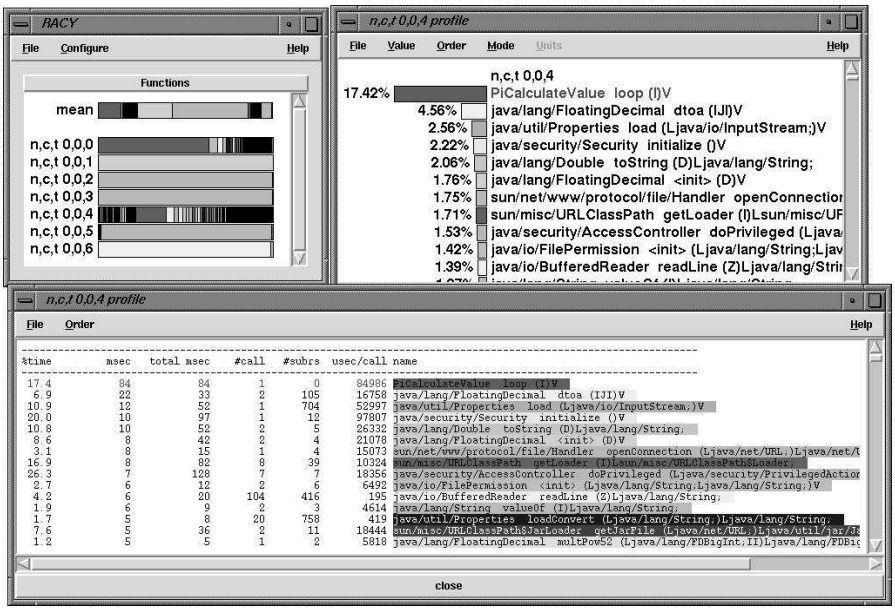

Figure 13. TAU RACY trace

In Figure 14 we can observe a Vampir trace with the different threads that the application are using. We can distinguish the moment which they were created and when have finished.

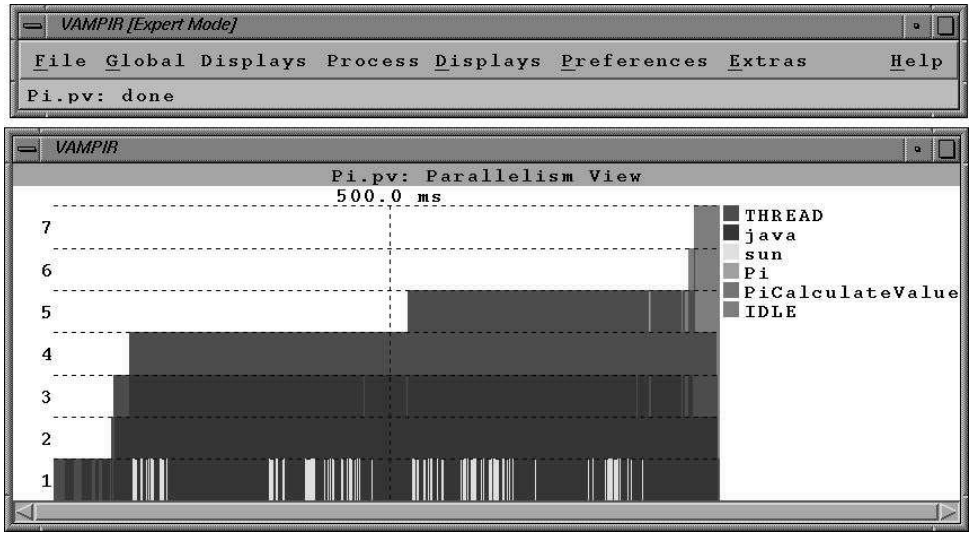

Figure 14. TAU Vampir trace

Strong points

- The visualization is relayed on other tools, probably more sophisticated
- It can profile MPI applications using mpiJava

Weak points

- The visualization tools are commercial and no open source.

More information about this profiler can be found in [12].

5. non-JVMPI compliant tools

Once we have shown a representative group of JVMPI compliant profilers, we will show some of the existing Java profilers that don't use JVMPI as the instrumentation technique. The main feature that distinguish these profilers from the JVMPI compliant are the much more specific

information that can be collected. This fact is possible because they can instrument the JVM in the way they want.

5.1. JaViz

The project of JaViz started in 1997 at the University of Minnesota with the idea of providing Java software developers an easy way to collect performance data and analysis. Apart from the typical problems of performance in Java applications, as inefficient methods, excessive synchronized methods or horrible memory management, JaViz is able to trace distributed applications using RMI (Remote Method Invocation) and show the point where the distributed application behavior is worse in a single trace. The steps covered by this profiler are:

- **Instrumentation**: JaViz uses an instrumented JVM capable of collect information about start and exit time of methods and to record client/server interactions. JaViz uses instrumentation as the profiling technique, in other words, adds code to the JVM directly to obtain information about each method's execution. The information obtained by JaViz can also be obtained by JVMPI although the time precision descends to milliseconds (JaViz has a microsecond precision in timestamps). Although they comment that are trying to use JVMPI as much as possible to remove JaViz's dependence over a modified JVM. Anyway, the tracing of client/server activity still need the modifying RMI library implementation.
- **Performance data extraction**: When a program is executed with the instrumented JVM three trace files are generated:
  - ✓ *.prf*: Contains detailed information of the execution for example, the call and return timestamps for every method executed.
  - ✓ *.clp*: Contains information about all outgoing RMI calls, also called server profile.
  - ✓ *.svp*: Contains information about all ingoing RMI calls, also called client profile.

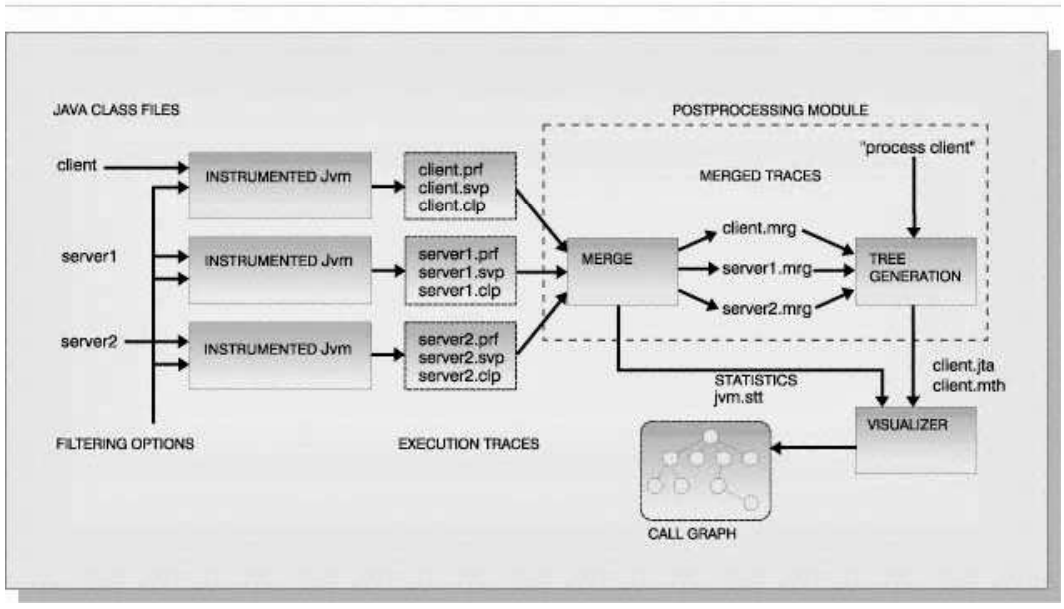

Figure 15. JaViz profiling process

There are some filter options that helps to control the amount of data collected, for example you can eliminate the java libraries methods profiling. Once the three files are generated by the JVM, a post-processing must be done to create the trace file that will understand the visualizer. This post-processing consists of two steps:

- ✓ *merge step*: The client and the server profile are merged in one file interlacing the client and server RMI calls. The main part consists of linking clients calls recorded in one trace file with the appropriate entry in the server profile.
- ✓ *tree generation*: This step generates the form like the information will be presented by the visualizer. The main goal of the tree generation step is to make the parent-child relationship among the different methods existing in the merged trace file.

- **Visualization**: JaViz has a visualizer that presents the information in an execution tree, with callers being parents of callees. This tree will include the execution of all threads and the interaction between different JVMs. The visualizer shows the tree with detailed node information, the user can navigate through the tree, expand and contract parts of the tree, search for a method, etc. Figure 16 shows the visualization window. It consists of two windows, the tree window and the node information window. The first shows the tree itself and the second displays the detailed information of the node selected in that moment.

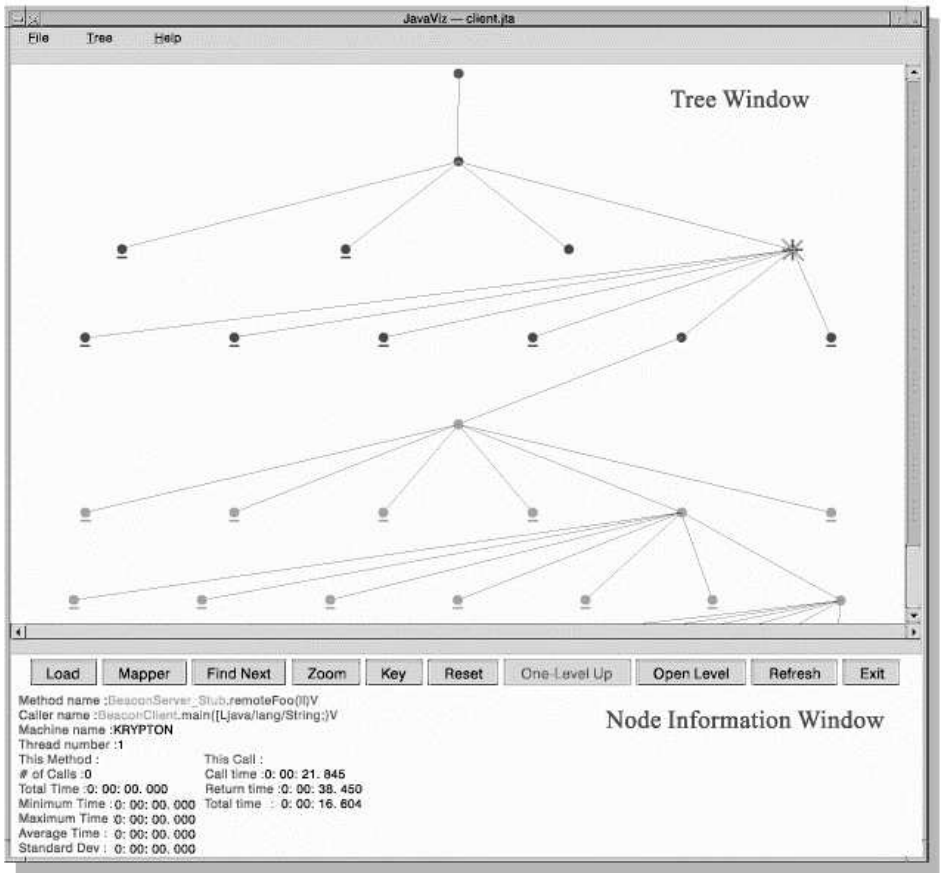

Figure 16. JaViz visualization window

Strong points

- It supports RMI tracing

Weak points

- Execution time is increased considerably (50 %) when the application is traced
- The merge process is very slow for a 17 Mb trace file it takes 30 minutes in a Pentium II 400 Mhz.
- The use of an instrumented JVM

More information about this profiler can be found in [4].

5.2. Jinsight

Jinsight is a profiler developed by IBM to show performance bottlenecks, object creation, garbage collection, thread interaction, deadlocks and program execution patterns. The Jinsight Tech preview is included with Visual Age for Java for Windows, or you can download the Jinsight for Windows 95/NT and for AIX.

The steps covered by this profiler are:

- **Instrumentation**: Jinsight offers an instrumented JVM with which the application must be executed to obtain the profile. The actual version comes with an instrumented JVM based on JDK 1.1.8.
- **Performance data extraction**: All the performance data extraction is done inside the instrumented JVM. When a java application is executed with the instrumented JVM it produces a Jinsight trace file. You can choose among options to turn tracing on and off, limit the type of information recorded and to mark significant event in the trace file.
- **Visualization**: Jinsight incorporates a visualizer to present the Jinsight trace files, it can show this information in different views:
  - ✓ *Histogram view*: Shows the program's use of resources, mainly CPU and memory. As we can see in Figure 17, each row represents a class followed by rectangles representing the instances of each class, colors can be selected according to various criteria. Clicking an individual element you can get more detailed information of that element. The lines between instances represent the relationship between objects.

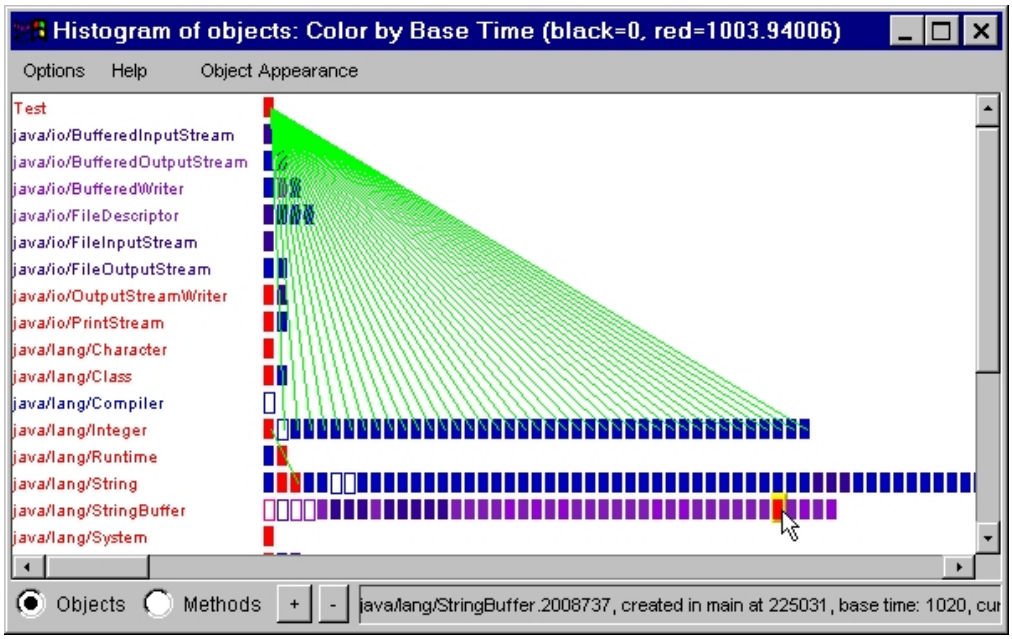

Figure 17. Histogram View

✓ *Execution view*: It shows the program execution sequence. As we can see in Figure 18, each of the four vertical lines shows a thread, time proceeds from top to bottom and each colored stripe shows the execution of a method on an object, the stripes are color-coded by class.

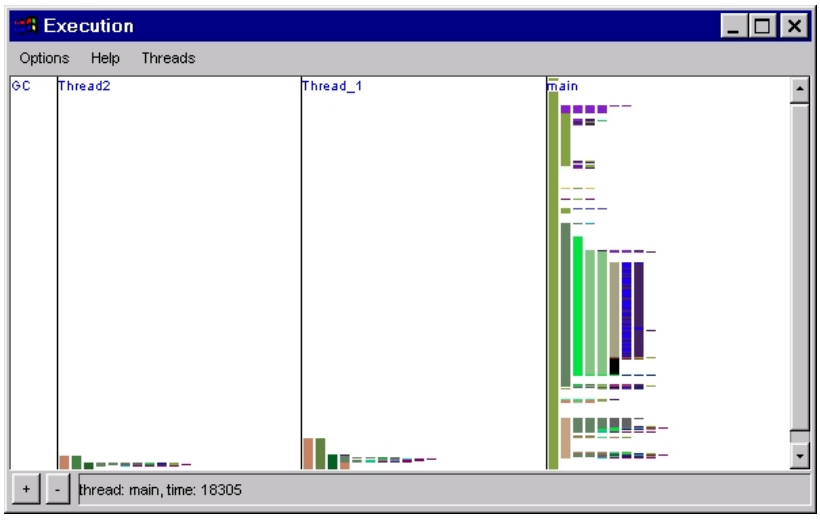

Figure 18. Execution View

A new feature incorporated in the version 2.0, is the ability to define slices of the trace information corresponding to features of your run you would like to study. Moreover you can define workspaces that give you a working context containing a number of views and slices that group information within a focus.

Strong points

- Good and original visualizer
- Easy to understand the way the methods behave

Weak points

- The size of trace files is huge 15-30 Mb per minute of execution
- The visualizer reads quite slow the trace taking to consideration the size that the trace file could have. (12 Mb per minute of the trace file in a Pentium Pro at 200 Mhz.)
- The use of an instrumented JVM

More information about this profiler can be found in [5].

5.3. JIS

The Java Instrumentation Suite (JIS) is a set of tools designed to instrument Java threaded applications, avoiding the instrumentation and recompilation of the source code and/or the Java Virtual Machine. JIS is now in development at CEPBA and is only available for SGI Origin 2000.

- **Instrumentation**: The information is collected thanks to DITools [11]. DITools is a set of tools bringing an environment in which dynamically linked executables can be executed at runtime with unforeseen functionality, then code interposition is used by JIS to instrument the code of the JVM.
- **Performance data extraction**: JIS is capable to obtain the state of every thread in the JVM, the information relative to monitors in which threads are using, actions performed by a

thread like processor yield, entry/exit on a mutual exclusion, invocation of Java Thread API, etc. All this information is collected and is written in a Paraver trace format.

- **Visualization**: Paraver [8] is the tool used to show the information collected by JIS, this tool has been developed at CEPBA too. As we can see in Figure 19, it shows a line for each thread showing the state in a color-based code and express events with flags.

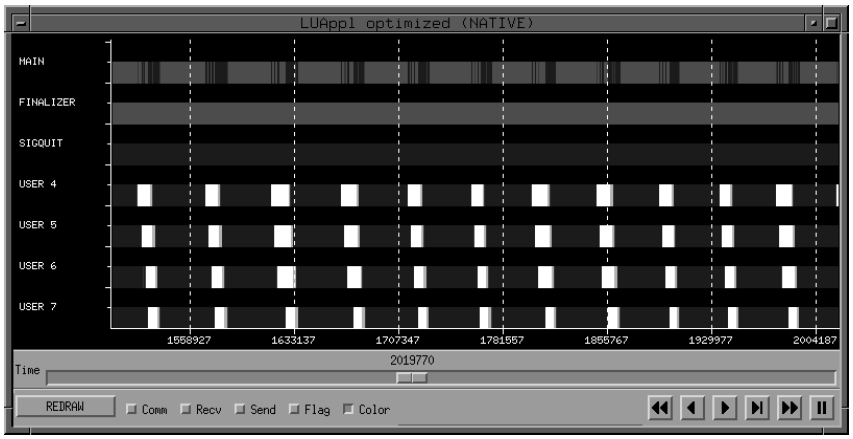

Figure 19. Paraver screenshot with a JIS trace

Strong points

- It can obtain some specific information that other tool cannot get
- Using Paraver as a visualization tool, it can get all the analysis facilities that this tool has

Weak points

- DITools and the JIS only work on the SGI Origin 2000
- User methods or information is not collected

More information about this profiler can be found in [2].

6. Conclusions

The most remarkable fact is the number of profilers that support JVMPI in comparison with the non-JVMPI profilers. It seems that JVMPI has been accepted by the developer community to construct general-purpose Java profilers. We will try to understand why this happens.

First of all, most of the information generated by non-JVMPI profilers can be generated using JVMPI (object operations, class operations, thread-aware, monitor synchronization), except from RMI support. This demonstrates that the information collected by JVMPI is general enough to generate general-purpose profilers.

However, profilers don't have to rely on JVMPI only, more specialized profilers may need some more specific information, then another techniques or tools should be used in the instrumentation or maybe the use of an instrumented JVM.

Another concept remarkable is the effort to establish JVMPI as a standard to be used by all profilers. This may provoke that when a new version of JVM appear if the JVMPI API is maintained no changes will need the profiling tools.

On the other hand, the profiler company of a non-JVMPI profiler has to instrument the new JVM. Moreover internal changes in the new JVM can provoke great changes in the profiler or re-designing parts of it, etc.

The company that wants to construct a Java profiler using JVMPI have to dedicate their effort in thinking of the way they are going to show the information and decide from the

information generated by the JVMPI which part are interesting to them. This reduces costs to construct a java profiler.

One of the main drawbacks of JVMPI is the intrusion provoked when an application is being profiled with JVMPI. All the JVMPI profilers don't talk of the intrusion provoked by their instrumentation, though non-JVMPI profilers as JaViz are as intrusive as JVMPI are.

Finally we will conclude that JVMPI is a very powerful tool to construct profilers in Java, although some improvements could be done like:

- Reduce the intrusion during profiling
- RMI support

Acknowledgements

This work has been supported by Ministry of Education of Spain under contract TIC98-511 and by Direcció General de Recerca of the Generalitat de Catalunya under grant 2001FI 00694 UPC APTIND. We acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying the human and computing resources for our work. We acknowledge also the help given by Juan A. Perez, Eduard Ayguadé and Jordi Torres.

References

[1] http://www.cepba.upc.es/bjs

[2] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. Proc. of the Second Annual Workshop on Java for High-Performance Computing, ICS00. St. Fe, New Mexico, May 7, 2000. http://www.ac.upc.es/torres/BJS/jhpc00.pdf

[3] HAT: Heap Analysis Tool http://java.sun.com/people/billf/heap/index.html

[4] H. Kazi. JaViz: A client/server Java profiling tool. IBM System Journals. http://www.research.ibm.com/journal/sj/391/kaziref.html

[5] Wim De Pauw, G. Sevitsky, E. Jensen. Jinsight: A tool for visualizing the execution of Java programs http://www-106.ibm.com/developerworks/library/jinsight/

[6] jProf profiler http://starship.python.net/crew/garyp/jProf.html

[7] Java Virtual Machine Profiler Interface (JVMPI) http://www.javasoft.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html

[8] 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

[9] OptimizeIt! The ultimate Java performance profiler http://www.optimizeit.com/

[10] Nathan Meyers. PerfAnal: A Performance Analysis Tool http://developer.java.sun.com/developer/technicalArticles/Programming/perfanal/index.html

[11] A. Serra, N. Navarro, T. Cortes. DITools: Application-level support for Dynamic extension and flexible composition. In proceedings of the USENIX Annual Technical Conference, June 2000 http://personals.ac.upc.es/alberts/fpc.html

[12] TAU: Tuning and analysis tools http://www.cs.uoregon.edu/research/paracomp/proj/tau/

[13] D. Viswanathan, S. Liang. Java Virtual Machine profiler interface. IBM System Journal, Vol. 39, No. 1, 2000