Tracing methodologies and tools for AI and Data Mining JAVA frameworks

Stagi, Roberto
Supervisors: Gil Gomez, Marisa (UPC); Garza, Paolo (PoliTO)

· Universitat Politècnica de Catalunya · Politecnico di Torino ·
· Barcelona Supercomputing Center ·

Abstract. Tracing is an important tool in multi-threaded applications. The tools from the BSC were not designed to be suitable with complex JAVA applications. In this study, these tools have been upgraded to trace more kinds of events from the JVM, and they are applied to some known AI and Data Mining frameworks. The application is focused on revealing some key events to better understand the behaviour of such applications.

1 Introduction

The present report is the first one for my thesis Project. The general intent of these reports is to inform about the progress done. For this first one, a background information about the thesis purpose, the tools used and the environment is given. For the next ones, I'll tend to assume this background for granted for who reads.

2 Objective

The final purpose of the thesis is to find some patterns and methodologies in the analyzing Java applications focused on Artificial Intelligence and Big Data. The importance of such analysis is basically the need to "measure to improve".

Searching on-line, it appears quite easily how tough it is to find good sources for this kind of work. It is in the sight of this interest that the present thesis is developed. Associated with the thesis, there's an internship at the Barcelona Supercomputing Center (abbreviated as BSC). The BSC provides some tools to perform this kind of analysis, but they are not suited for analyzing Java applications. For this reason, the interest by the BSC is to improve such tools and to gain the right experience in analyzing not only Java applications, but the above mentioned kinds of applications.

3 Environment

In brief, the work environment is composed of the tools from the BSC, installed on a Docker machine for portability reasons. The different tools and the setup is described below.

3.1 Paraver

Paraver is developed to visually inspect the behaviour of an application and then to perform detailed quantitative analysis. It has a clear and modular structure, which gives to the user expressive power and flexibility.[1]

The Paraver trace format has no semantics. Thanks to that, supporting new performance data or new programming models requires only capturing such data in a Paraver trace. Moreover, the metrics are not hardwired on the tool, but programmed. To compute them, the tool offers a large set of time functions, a filter module, and a mechanism to combine two time lines.

Preserving experts knowledge is made very simple. In fact, any view or set of views can be saved as a Paraver configuration file, which reduces the recomputing of the view with new data to a simple loading of a file.

3.2 Extrae

Extrae is a tool that injects probes into the target application, in order to gather information regarding the application behaviour and performance. It uses different interposition mechanisms to inject the probes, and generates Paraver tracefiles for an analysis carried after the termination of the program. [2]

The most common way to inject probes is through the "*LD_PRELOAD trick*". That is setting the `LD_PRELOAD` environment variable to the path of the Extrae tracing library, which makes Extrae to be loaded before any other library. In this way it will be able to implement some wrappers for some functions of the known programming models, being able to inject the various probes necessary to gather the valuable information. This is the same way used in this thesis.

Extrae is subjected to different settings. In order to facilitate the configuration, it can be configured through an XML file, normally specified in the environment variable `EXTRAE_CONFIG_FILE` before executing the target application.

For the specific case of Java, Extrae used to offer a basic thread detection based on pthread (each Java thread is mapped to a pthread in Linux). Moreover, it was implemented an API to manually set the events from some Java code, and a feature to automatically trace some functions by generating aspects used by AspectJ. The latter feature was not working and has been fixed. More information in later sections.

3.3 Docker

Docker is a set of services that use OS-level virtualization to deliver software in the so-called *containers*, isolated from the other parts of the file system. [3]

In this case, the container is an image based on OpenSUSE with the tools of the BSC installed and ready to be used.

The reason behind choosing OpenSUSE is because the MareNostrum 4 (the BSC supercomputer) is based on the SUSE Operating System, of which OpenSUSE — being its free version — is the most similar OS that could be easily installed on a Docker image.

3.4 GitHub

All the files, including source code, scripts and installation files, are stored in a git repository on my personal GitHub account. The link is the following:

`https://github.com/rstagi/thesis`

The repository, inside a sub-folder named "*installation*", contains a git submodule which points to a fork of the original Extrae repository. In the fork repository, under a branch named `javatrace`, there are all the changes that I'm making in order to extend Extrae's functionalities, making it more suitable to trace Java applications.

Moreover, in the root folder there are the *Dockerfile* for the Docker image and the scripts to build the image and running the applications, while in the *installation* folder there are the Paraver and AspectJ installation files. Running the application is done inside the Docker container, by using its installed software for the generation and visualization of the traces. To test such scripts, there is an "*examples*" folder with an example of Java application and a *Makefile* to build it and run it within the container.

Finally, a README file is provided to help in using the scripts and the examples, as well as a folder named "*docs*" which contains this, the following reports and presumably the various future thesis drafts.

3.5 Examples of usage

Set-up To set up the environment, it is necessary to clone the repository together with the submodule. This can be done by running:

```
git clone https://github.com/rstagi/thesis.git --recursive
```

Docker image build To build the Docker image, it is provided a script named `build_docker_javatrace.sh`, which can be found in the root directory of the repository and needs to be run there:

```
./build_docker_javatrace.sh
```

This command will at first clean the Extrae submodule (by running a `git clean` command inside it) and then it will build the Docker image by following the Dockerfile.

An option `-pull [<target_branch_or_tag>]` can be specified, in order to switch on a different branch. Normally, the submodule will point to the last commit associated with the current commit of the "thesis" repository. By adding this option, the script will switch on the target branch or target if specified, or to `javatrace` if no target has been specified after `-pull`.

The image will be based on OpenSUSE. During the building process, the necessary tools (like compilers, developers libraries, etc.) will be installed, the installation files of Extrae (installed by using the sources), Paraver and AspectJ will be copied inside and used, and finally the image will be tagged as `extrae/javatrace`.

Running the program To run and trace a Java program, there is the script `run_javatrace.sh`. This can be used as the following:

```
./run_javatrace.sh [-jar] [-r, -R] [-show] <target>
```

The `target` can be either a Java class or a jar file. For the former, which was the standard mode in which Extrae used to work with Java programs, the class should be present in the directory where the command is launched or in the classpath. In both cases, all the files in the target's directory will be copied inside the Docker container. The solution is not very elegant, but it's effective and it makes the logic behind the script much easier. For the purposes of the analyzed applications it's quite enough. The options are not mandatory, and they have the following meaning:

- `-jar`: the target is a jar file
- `-r/-R`: the files will be copied in recursive mode
- `-show`: once the program terminates, it will automatically run Paraver to show the trace

Show the traces To show the traces generated by a program, in addition to the `-show` option of the running script, a new script, named `show_trace.sh` is provided:

```
./show_trace.sh <target_prv_file>
                [-conf <pcf_file>]  [-row <row_file>]
```

It takes a `prv` file as input. Optionally, a configuration and a row files can be specified using the options `-conf` and `-row`. The configuration file (with extension `.pcf`) is a file with some saved settings from previous traces analysis (it is already described in the Paraver subsection). The row file, instead, contains other metadata such as tasks/threads names, functions names, etc.

All the input files are copied inside the container. Paraver is then executed receiving them as input files.

Examples As previously stated, there are some examples in the repository. At the moment of writing, they are just two: one with a jar file and one using the classes.

In both cases, the Docker image needs to be built first. Moreover, there are the *Makefile*s to help in running the examples using the Docker container. The useful targets are the following:

- `make run`: runs the program inside the container and copies the files in the current directory
- `make runshow`: runs the program as the `run` target and also shows the traces using Paraver
- `make show`: needs to be used after the `run` or `runshow` targets and shows the output trace in Paraver

4 What has been done

Until now, the work has been focused on understanding the tools from the BSC and starting to improve them to correctly gather information from the Java applications.

Moreover, the environment has been set up like described in the previous section.

4.1 Improvements to Extrae

The changes to the BSC tools actually involve just Extrae. As stated in previous sections, the idea of Paraver is to keep it like it is and changing just the methods for gathering traces from new kinds of applications or programming models. Keeping this in mind, I worked on Extrae interfacing it with the JVMTI, that is the "Java Virtual Machine Tool Interface".

In addition, I developed a feature to let the user analyze jar files –and not only java classes in the Classpath– by adding the `-jar` option to `extraej`, that is a bash script installed among the Extrae packages which is responsible to trace Java programs.

From Figure 1, we can see some traces which refer to the execution of PiExample, the Java example inside Extrae main project (example/LINUX/-JAVA/automatic). In this example, the main thread calculates the value of π by using at first only 1 thread, then 2 threads, 4 threads and 8 threads. This behaviour is verifiable from the traces represented in the figure.

From 1a, we can see how before the changes the traces had just running and fork events. In theory, the main thread (the second one) should be always waiting for the others. This behaviour, is verifiable in 1b, where the traces are the same but with an extended event of synchronization. Such event has been traced using JVMTI events. Zooming on the trace, we obtain the trace in 2a, where you can see that there are very small running pieces of trace just after the thread pool ends and just before forking again. Both in 1a and 1b, the program is traced by using pthreads. For this reason, the detected threads are more than we expected, because of some internal threads management by the JVM. In every program I tested, using pthreads, there were always these 20 threads (from *Thread 1.1.3* to *Thread 1.1.20*) which are always almost the same for every program. Apparently, they are not useful at all to understand the program behaviour (also for another reason explained soon).

To switch to a better tracing methodology, I decided to use the JVMTI events. In the callbacks of such events, I placed a probe to trace such threads and their waiting states. New events values have been defined, and the 20 threads traced in the other method using pthreads are not traced anymore (which is another index of their pointlessness for the purposes of the execution). In the same way, a new thread is appeared, which is the *DestroyJavaVM* thread. Such trace, can be seen in 1c, and its zoom can be seen in 2b.

In addition, another change made is about thread names. Now the name of the thread is taken from the name given by the JVM.

In addition to these Extrae's core changes, an option to `extraej` has been added. Until now, `extraej` was designed to just trace classes present in the Classpath or in the working folder where it was launched. Now, an option `-jar` has been added in order to consider the target as a jar file. Execution and tracing remained exactly the same.

N.B.: The following subsections will be dedicated to technical implementation details. To have just an overall idea of how the works is being developed and how to use the updated software, it is not necessary to read them.

4.2 JVMTI Events to trigger Tracing Probes

Before my intervention, the JVMTI was used just to trace information about object allocation. Information about threads were taken using pthreads probes. Indeed, each Java threads is mapped using a pthread in Linux. Unfortunately, the only information given by this method were the start and the end times of the threads. All other valuable information (like synchronization, forks and so on) were not traced at all, because the JVM does not use pthreads' functions to perform such activities.

Operations like thread creation and synchronization, instead, are tracked by JVMTI events. In this way, I could inject probes in some callbacks for these events, resulting in an efficient tracing of all the above information.

Going more deeply, from the technical point of view the part of code responsible for this tracing starts like the following:

Code 1.1: JVMTI Agent init

```
1  JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void
       *reserved)
2  {
3      jvmtiEnv          *jvmti;
4      jvmtiCapabilities capabilities;
5      jvmtiEventCallbacks callbacks;
6
7      /* Get JVMTI environment */
8      (*vm)->GetEnv(vm, (void **)&jvmti, JVMTI_VERSION);
9
10     /* Get/Add JVMTI capabilities */
11     memset(&capabilities, 0, sizeof(capabilities));
12     capabilities.can_generate_garbage_collection_events = 1;
13     capabilities.can_generate_exception_events = 1;
14     capabilities.can_tag_objects = 1;
15     capabilities.can_generate_monitor_events = 1;
16     (*jvmti)->AddCapabilities(jvmti, &capabilities);
17
18     /* Set callbacks and enable event notifications */
19     memset(&callbacks, 0, sizeof(callbacks));
20     callbacks.<JVMTI_event_callback> = &<custom_callback_function>;
21     (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
           <JVMTI_Event_ID>, NULL);
```

```
22 }
```

The callbacks will just call the probes, which will then use Extrae's core functions to trace the right events.

Code 1.2: Example of callback

```
1  static void JNICALL Extraej_cb_Thread_start (jvmtiEnv *jvmti_env,
2      JNIEnv* jni_env, jthread thread)
3  {
4
5      jvmtiThreadInfo ti;
6      jvmtiError r;
7      UNREFERENCED_PARAMETER(jni_env);
8
9      if (thread != NULL)
10     {
11         r = (*jvmti_env)->GetThreadInfo(jvmti_env, thread, &ti);
12
13         //Apparently, when thread is ending, ThreadStart is called with
               thread_group=0
14         if (r == JVMTI_ERROR_NONE && ti.thread_group){
15             Extraej_NotifyNewThread();
16
17             unsigned threadid = THREADID;
18             if (strcmp("", Extrae_get_thread_name(threadid)) == 0)
19                 Extrae_set_thread_name(threadid, ti.name);
20
21             Extrae_Java_Thread_start();
22         }
23     }
24
25 }
```

Where `Extrae_Java_Thread_start()` is the actual probe, implemented as follows:

Code 1.3: Example of Probe

```
1  void Extrae_Java_Thread_start(void)
2  {
3      if(! EXTRAE_ON()) return;
4
5      if (EXTRAE_INITIALIZED() && !Extrae_get_pthread_tracing())
6      {
7          Backend_Enter_Instrumentation ();
8
9          DEBUG
10         TRACE_MISCEVENTANDCOUNTERS(LAST_READ_TIME, JAVA_JVMTI_THREAD_EV,
               EVT_BEGIN, EMPTY);
11         Extrae_AnnotateCPU (LAST_READ_TIME);
```

```
12
13          Backend_Leave_Instrumentation ();
14      }
15  }
```

One main concern about using JVMTI events, was the problem in identifying the threads. Indeed, Java does not provide a thread identifier through JVMTI or its APIs. However, as we stated earlier, each Java thread is created using pthreads. For this reason, pthread identifiers are going to be used to identify the threads inside Extrae's core. The identifier is actually automatically taken from the pthread identifier, without adding anything in any source file.

The list of the modified files is the following:

- ◇ config/java.m4
- ◇ src/common/events.c
- ◇ src/common/events.h
- ◇ src/java-connector/jvmti-agent/extrae-jvmti-agent.c
- ◇ src/launcher/java/Makefile.am
- ◇ src/merger/paraver/java_prv_events.c
- ◇ src/merger/paraver/java_prv_semantics.c
- ◇ src/tracer/wrappers/JAVA/java_probe.c
- ◇ src/tracer/wrappers/JAVA/java_probe.h

4.3 New Extraej feature

Execution traces of Java programs were only traced on Java Classes in the Classpath, or in the current folder where `extraej` was executed.

In order to make it easier (and cleaner), I added an option to analyse executions starting from jar files. Indeed, by using the `-jar` option, `extraej` will consider the target as a jar file. The change is as simple as the following piece of code:

Code 1.4: extraej.bash.skeleton

```
167 elif [[ "${1}" = "-jar" ]]; then
168     jarmode="yes"

    ...

181 execute_java () {

    ...

186     options=""
187
188     if [[ "${jarmode}" = "yes" ]]; then
189         options=-jar
190     fi
```

And then, when executing the target:

```
201     ${JAVA} ${options} ${@}
```

...

```
212     ${JAVA} ${options} -agentpath:${EXTRAEJ_LIBEXTRAEJVMTIAGENT_PATH}
            ${@}
```

4.4 Bugfixes

At the moment in which I started with this work, there were some bugs in Extrae. These were all in the configuration file and in the `extraej` skeleton. Specifically, the problems were:

- ◇ AspectJ Runtime (`aspectjrt.jar` file) was missing from the Classpath
- ◇ AspectJ Weaver (`aspectjweaver.jar` file) reference was missing from Extraej skeleton

Both of them have been fixed.

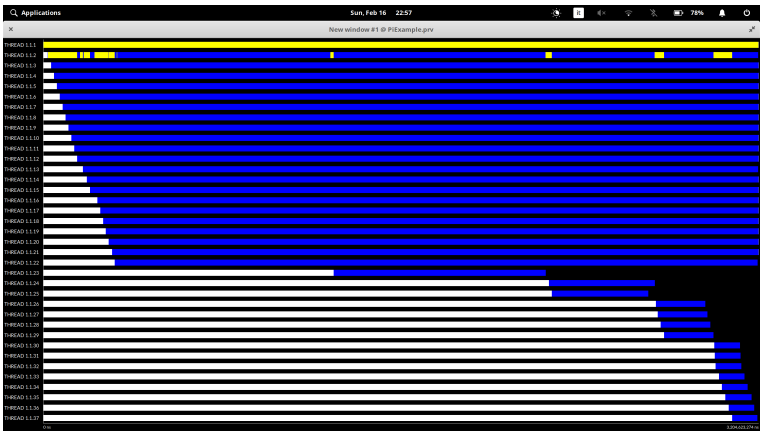
5 What to do from now

In order to get even better trace, I would like to proceed with the following changes in the tools:

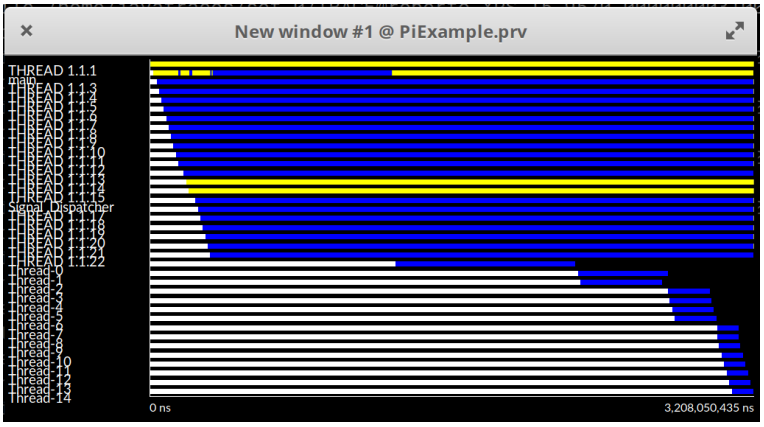
- ◇ Find a way to trace **inter-thread communications** (Paraver provides some graphical tools which efficiently represents such events);
- ◇ Trace **waits and idles**, which are still missing from Java traces.

Moreover, the following analysis should be carried out:

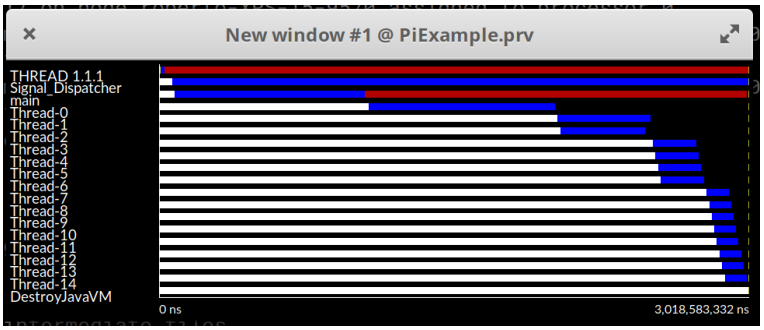
- ◇ **Test** the changes with some **Data Mining and AI frameworks**;
- ◇ **Test** the changes with **multi-process programs** (like Spark or Hadoop);
- ◇ As Marisa suggested, **test** the changes with **faulty applications** in order to discover some patterns.



(a) Before the changes

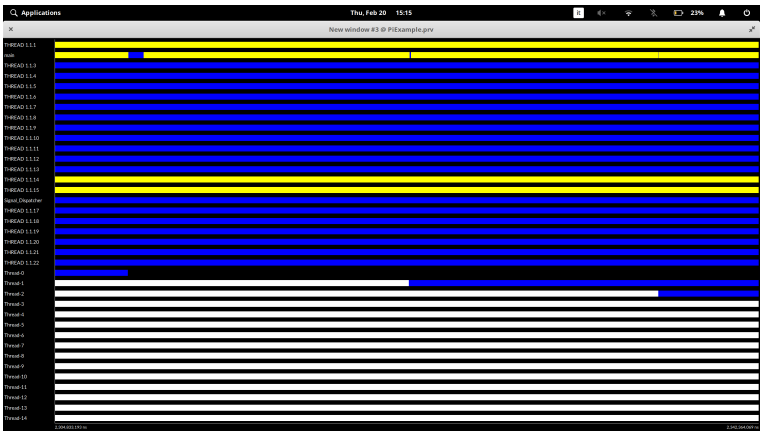


(b) After the changes, with pthread enabled

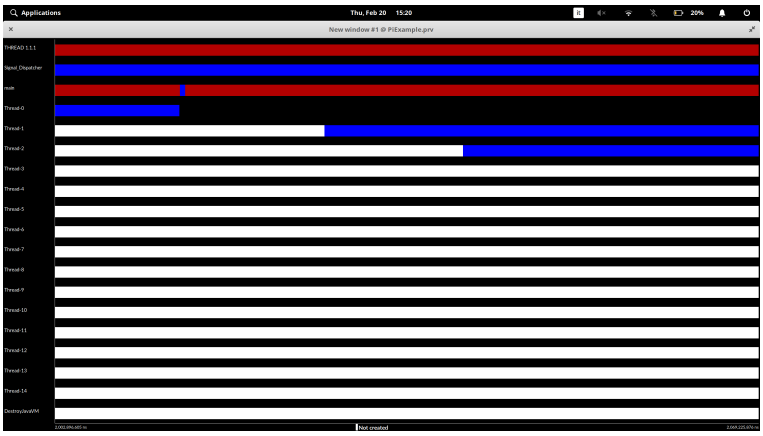


(c) After the changes, using jvmti events (that is with pthread disabled)

Fig. 1: Comparisons between before and after the changes. The executed program is the example from Extrae's project (PiExample).



(a) With pthread enabled



(b) With pthread disabled

Fig. 2: Zoom on the area between a thread end and a thread creation

References

1. (2020), https://tools.bsc.es/paraver
2. (2020), https://tools.bsc.es/extrae
3. (2020), https://en.wikipedia.org/wiki/Docker_(software)