POLITECNICO DI TORINO

Master's Degree in Computer Engineering
Data Science

- & -

UNIVERSITAT POLITÈCNICA
DE CATALUNYA (UPC) - BarcelonaTech

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

Master in Innovation and Research in Informatics
High Performance Computing

Developed during an Internship at the
Barcelona Supercomputing Center (BSC)

Master Thesis

Tracing methodologies and tools for Artificial Intelligence and Data Mining Java applications

Supervisors

Prof. Maria Luisa GIL GOMEZ[†]

Prof. Paolo GARZA[††]

[†]Department of Computer Architecture (DAC), UPC
[††]DAUIN, Politecnico di Torino

Author

Roberto STAGI

July 2020

Abstract

Acknowledgements

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

AOP

Aspect Oriented Programming

API

Application Programming Interface

BSC

Barcelona Supercomputing Center

HPC

High Performance Computing

JNI

Java Native Interface

JVM

Java Virtual Machine

JVM TI

Java Virtual Machine Tool Interface

Chapter 1

Introduction

1.1 Context: High Performance Computing, Artificial Intelligence and Java

Supercomputing and Artificial Intelligence are among the most important outcomes of the last decades. Both of them have been behind the scenes of many recent discoveries, correctly credited to other classes of instrumentation (e.g. the Hubble telescope), but that required supercomputing and AI as the enabling tools for large datasets processing—usually referred to as "Big Data" [1] [2].

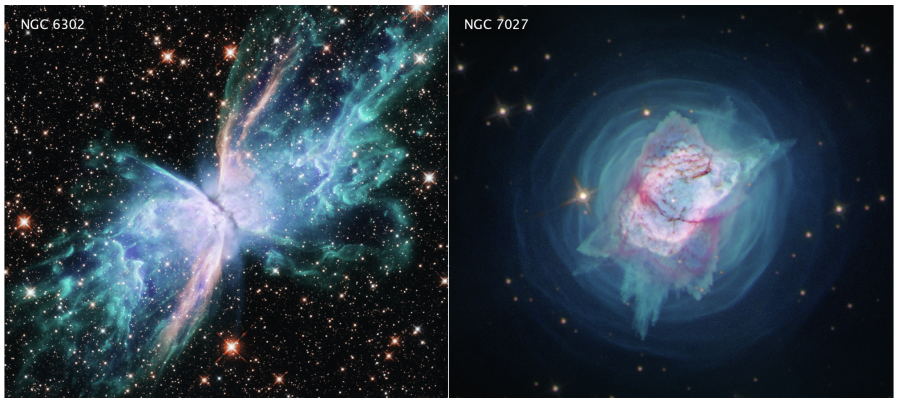

Figure 1.1: Two Planetary Nebulas photographed by the Hubble Telescope [3]

AI applications, from Deep Learning to Data Mining, going through Neural Networks and Clustering algorithms, together with most of the applications in general,

have been switching from a sequential paradigm to parallel and distributed approaches, that best fit the new hardware. The High Performance Computing (HPC) discipline is at the heart of these developments.

HPC is a field of endeavor that relates to all facets of technology, methodology, and application associated with achieving the greatest computing capability possible at any point in time and technology. The action of performing an application on a supercomputer is widely termed "supercomputing" and is synonymous with HPC (T. Sterling *et al* [1, p. 3]).

In this context, the Java programming language plays a marginal role. Languages such as R and Python are much more common when manipulation of Big Data and statistic analysis are the primary goals [4]. However, Java is still in high demand, it is employed in AI and runs effectively on supercomputers. Even if a smaller set of programmers use it for HPC applications, its influence in the AI world is not negligible and it deserves a larger attention to the tools that support its development in such environment.

1.1.1 Java-powered AI and Data Mining

The high and always increasing demand of AI features has affected almost all the programming languages. Research institutions and companies started to invest on AI and Machine Learning [5]. Java, as one of the most common languages, got a bunch of new libraries to enable the developers to access this various world, made of statistics and algorithms. Among all the frameworks for AI, Machine Learning and Data Mining, the ones listed below are probably the most common ones employed with Java. Worthy to be in a resume and capable of figuring in the skills requirement of some tech careers.

Weka The Waikato Environment for Knowledge Analysis (Weka) is an open source software developed at the University of Waikato, in New Zealand. The Weka workbench is a collection of machine learning algorithms and data preprocessing tools, providing a Java library and a graphical User Interface to train and validate data models. It is among the most common Machine Learning frameworks for Java, since it was one of the first ones and it is still maintained [6].

Apache Spark MLlib Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Spark Core provides distributed task dispatching, scheduling, and basic I/O functionalities, exposed through an application programming interface (for Java, Python, Scala, and R) centered on the Resilient Distributed Dataset (RDD) abstraction. RDD is

a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way [7]. Spark MLlib is a distributed machine-learning framework on top of Spark Core that implements many machine learning and statistical algorithms, simplifying large scale machine learning pipelines [8].

Apache Mahout Apache Mahout is a distributed linear algebra framework, written in Java and Scala, whose architecture is built atop a scalable distributed platform. Although Apache Spark is the recommended one, Mahout supports multiple distributed back-ends. The framework features console interface and Java API, that give access to scalable algorithms for clustering, classification, and collaborative filtering [9].

1.1.2 Distributed Java in HPC

The above frameworks are not thought for an HPC environment. The standard implementation of Weka, for example, is designed to run on standard machines (like PCs, laptops or small servers), with most of the algorithms implemented sequentially. This makes it difficult to gain advantage of a strongly parallel architecture like a supercomputer. Spark and Mahout, instead, run both on a distributed platform, which means that they're designed to run on a cluster of different machines instead of a unique system. Java is indeed perfectly suitable to work on a distributed environment, usually using frameworks like MapReduce[1], whose most common implementation is Apache Hadoop, written in Java.

Spark has its own core that work in a similar fashion, Mahout runs on a distributed backend and Weka too can go distributed with some packages, running on frameworks such as Spark or Hadoop [11]. All of them rely directly or indirectly on the map-reduce framework.

Although a supercomputer and a distributed environment, made of nodes connected over a network, are similar from the physical perspective, they're much different when speaking about logic organization. Nevertheless, a distributed system can be deployed on a supercomputer, by keeping the logic organization and taking advantage of its computational power. Such emulated environment can be reached with the use of software containers.

Software containers are a form of OS-level virtualization introduced by Docker. Even if an emulated distributed system can be made of Docker containers, the BSC

[1]MapReduce is a programming model for processing big data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a map procedure, which performs filtering and sorting, and a reduce method, which performs a summary operation [10].

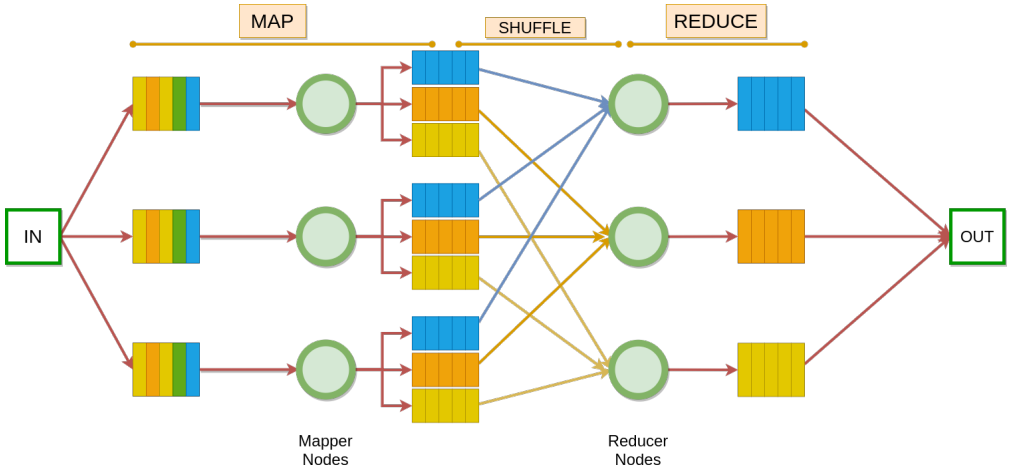

Figure 1.2: How the map-reduce framework works. The nodes can be virtualized using containers.

choice for MareNostrum is Singularity, another type of software containers created as an alternative to Docker for Clusters HPC environments[2].

Singularity enables users to have full control of their environment. Singularity containers can be used to package entire scientific workflows, software and libraries, and even data [12]. Distributed applications based on Hadoop or Spark can run on supercomputers, by using virtualized clusters made of Singularity containers.

1.1.3 Performance analysis

Quoting prof. Jesus Labarta, one of my professors at the UPC, «measurement techniques are "enablers of science". They are present everywhere, and so they are in Computer Science. In the specific case of HPC, they are represented by the performance analysis tools.»

Parallel program performance analysis and tuning is concerned with achieving efficient utilisation of system resources. One common technique is to collect trace data and then analyse it for possible causes of poor performance. The objective is to gather insights, both qualitative and quantitative, in order to increase predictability, build confidence and properly suggest improvements in the software [13].

A great way to describe the observed behaviour is through performance models, which take in consideration multiple model factors based on specific metrics (load

[2]Indeed, that is a common choice in most of the supercomputers worldwide.

balance, communication efficiency, etc.), making it easier to understand where to act.

At the BSC, performance analysis methodologies are mainly based on "traces", generated by instrumenting the libraries and taking several kinds of data from hardware counters, system calls or other low level mechanisms.

1.2 MareNostrum Tools Environment

The work in the present thesis is carried out as an intern at the Barcelona Supercomputing Center (BSC), that has one of the most powerful supercomputers in the world. It is situated right next to the Campus North of the UPC.

The BSC supercomputer is called MareNostrum (Figure 1.3), and it is at its 4th version. With 6,470.8 TFlop/s of max performance, at the time of writing the MareNostrum occupies the 30th position among the most powerful supercomputers in the world, according to the top500 list [14]. It has 165,888 cores, divided on 3,456 nodes, and counts more than 300 TB of memory [15].

Figure 1.3: MareNostrum 4 [15]

The MareNostrum software environment is based on the SUSE Linux Enterprise Server Operating System. There are several available modules ready to be loaded, and many of them are developed within the BSC research departments. Among these modules, we find the performance analysis tools developed at the Performance Tools Department of the BSC: Extrae and Paraver.

1.2.1 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.

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.

Paraver is not tied to any programming model, as long as the model used can be mapped in the three levels of parallelism expressed in the Paraver trace [16].

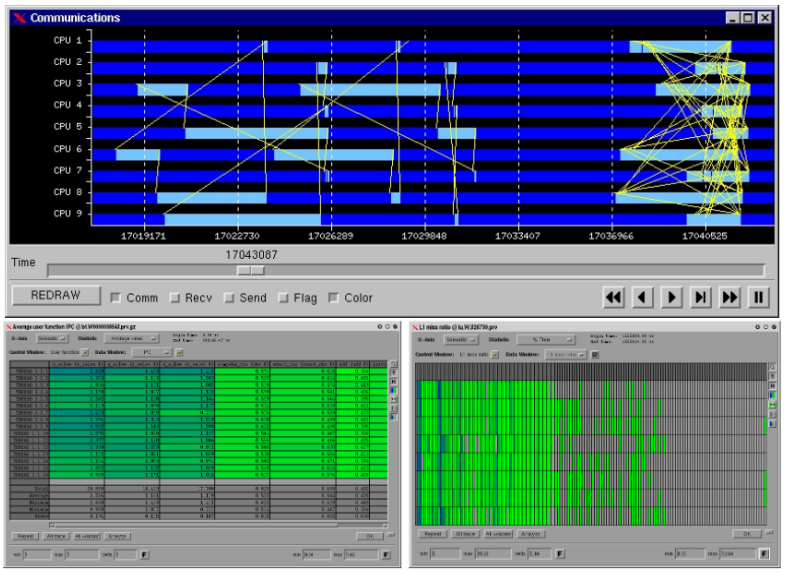

Figure 1.4: Example of traces analysis using Paraver, taken from the BSC website [16]

A Paraver trace file has three types of records: state, event and communication. The state is a record associated to a thread for a specific time interval (running, idle, synchronization, communicating, etc.). The event record is a punctual event that happen on one object, encoded in two integers (type and value). Finally, the communication event relates two objects in two points in time, representing a pair of events in two different threads and a relationship between the event happening in the first thread and the second [17].

The Paraver trace file is a set of three textual files containing the application activity (.prv), the labels associated to the numerical values (.pcf) and the resource

usage (.row). Currently, there are several ways to generate Paraver trace-files, like Extrae, Dimemas or other Translators (Figure 1.5) [18]. The tool on which this thesis will focus is Extrae, the core of the instrumentation package developed at the BSC.

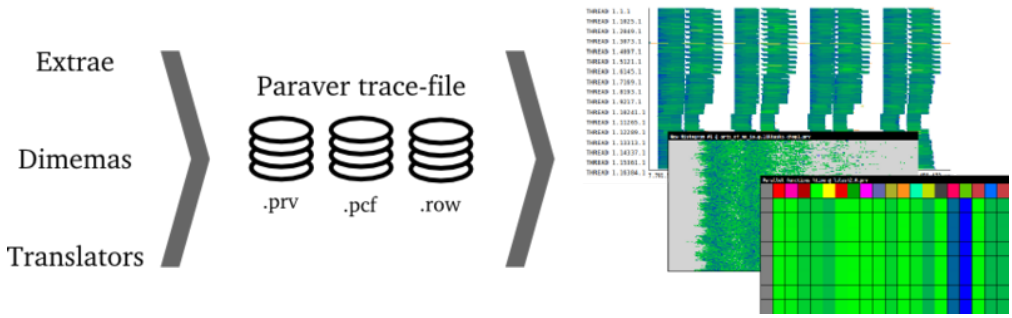

Figure 1.5: Paraver traces generation, taken from the BSC website [19]

1.2.2 Extrae

Extrae is a tool that uses different interposition mechanisms to inject 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 trace-files for an analysis carried after the termination of the program [20].

The most common interposition mechanism consists in using the Linker Preload, that expects to set the `LD_PRELOAD` environment variable to the path of the Extrae tracing library before executing the target program. In this way, if the pre-loaded library contains the same symbols of other libraries loaded later, it can implement some wrappers for the functions valuable to get data from. This is what Extrae does for most of the instrumented programming models.

Another mechanism consists in manually inserting some probes. On top of the instrumented frameworks, Extrae provides an API which gives the user the possibility to manually instrument the application and emit its own events.

Extrae is subjected to different settings, configured through an XML file specified in the `EXTRAE_CONFIG_FILE` environment variable. These settings have the control on almost everything in the instrumentation process. They can enable the instrumentation of some libraries instead of others, the hardware counters, the sampling mode, etc. The tracefiles generated by Extrae are formatted to comply with the Paraver trace format.

Extrae supports different programming models (like MPI, OpenMP, CUDA, etc.)

and also basic threads instrumentation and events tracing in Python, Java and C/C++ (instrumenting the pthread library, in addition to the above mentioned programming models).

The instrumentation for Java, which is the main focus for this thesis, is poorly implemented. It supports just a basic thread instrumentation, by tracing only the Running state, the Exception event and the Garbage Collection event. The features currently implemented for Java instrumentation are explained in chapter 2.

1.3 Problem Statement and Goal

At the BSC, tracing a Java program is something that has not been given much attention. In the BSC performance tools, Java is poorly instrumented, relies on just few events that do not give enough valuable information on what the program does and how it is improvable.

The contributions of this work aim to:

- Design and implement a Java instrumentation platform, on top of one of the BSC performance tools: Extrae;
- Find some patterns and methodologies that can help analysing Java applications in an HPC environment;
- Doing so, with keeping a focus on the AI frameworks, which represent a large portion of the Java applications that typically run on supercomputers.

Since I developed this thesis as an intern at the Barcelona Supercomputing Center (BSC), the starting basis for the work will be the BSC performance tools. The interest for the company—and so my job as an intern—was to improve such tools (Extrae, mainly) and to gain the right experience in analyzing the typical HPC applications implemented in Java.

In order to do so, the thesis will be developed starting from an analysis of the state of the art of Extrae for Java (chapter 2), by instrumenting an example multi-threaded program. It follows a discussion about the different possibilities of instrumentation methodologies (chapter 3), before actually implementing a solution that tries to trace the Java threads and related events (chapter 4 and chapter 5), in order to get some traces detailed enough to properly study the behaviour of an application.

Finally, the implementation is extended and applied to a case study: the Hadoop framework (chapter 6). The study ends with a discussion of the results (chapter 7) and a conclusion (chapter 8), which wrap everything up and give a sense to what

has been done and what can be improved.

1.4 Materials and Methods

If not specified, all the executions reported in this thesis ran on a Dell XPS15 9570, with an Intel i7-8750H CPU at 2.20GHz and 32GB of DDR4 RAM. The programming tools employed in the work have been the following:

- Visual Studio Code for all the `C`/`C++` code
- Jetbrains IDEA IntelliJ for all the Java/AspectJ code
- Docker to virtualize the OS and the tools environment
- GitHub to store all the code and manage Extrae's pull requests and versioning

All the examples are easily reproducible thanks to the virtualization offered by the Docker image. The image is based on OpenSUSE, provides the BSC tools (Extrae and Paraver) and all the other dependencies (AspectJ, Java, etc.) are installed and ready to be used. The reason behind choosing OpenSUSE is because of the MareNostrum 4 Operating System (SUSE Linux Enterprise Server), of which OpenSUSE—being its free version—is the most similar OS that could be virtualized with a Docker image.

An explanation of the usage of the environment, including the steps to build and setup the image, as well as the instructions on how to run the examples, is present in appendix A.

Chapter 2

Extrae for JAVA: State of the Art

This chapter will go through the state of the art of Extrae's instrumentation for Java. It is not totally absent, but its tracing capabilities for Java are currently poorly implemented. The discovery of the available features will be carried out by looking at one at the Extrae's Java example, provided in the Extrae package, which calculates the π number with a parallel algorithm. By analyzing it, this chapter will cover all the aspects of how Extrae traces the Java threads: how it gathers the data and spot the events, how it is launched and what are the experimental features that were added. Additionally, the generated traces will be analyzed in order to understand what events are missing.

2.1 The example program

The program that is going to be analyzed is a simple algorithm to calculate π. It does so 5 times: the first time with a sequential algorithm, the other four with a parallel implementation, respectively with 1, 2, 4 and 8 threads.

The main class is shown in Code 2.1[1]. A scheme of the expected behaviour is reported in Figure 2.1.

Code 2.1: PiExample.java

```
1 public class PiExample
```

[1]In reality, that is just an extract of the real main class. The real one contains some time measurements and print messages.

```
{
  public static void main (String [] args)
  {
    PiSerial pis = new PiSerial (steps);
    pis.calculate ();

    PiThreaded pit1 = new PiThreaded (steps, 1);
    pit1.calculate ();

    PiThreaded pit2 = new PiThreaded (steps, 2);
    pit2.calculate ();

    PiThreaded pit4 = new PiThreaded (steps, 4);
    pit4.calculate ();

    PiThreaded pit8 = new PiThreaded (steps, 8);
    pit8.calculate ();
  }
}
```

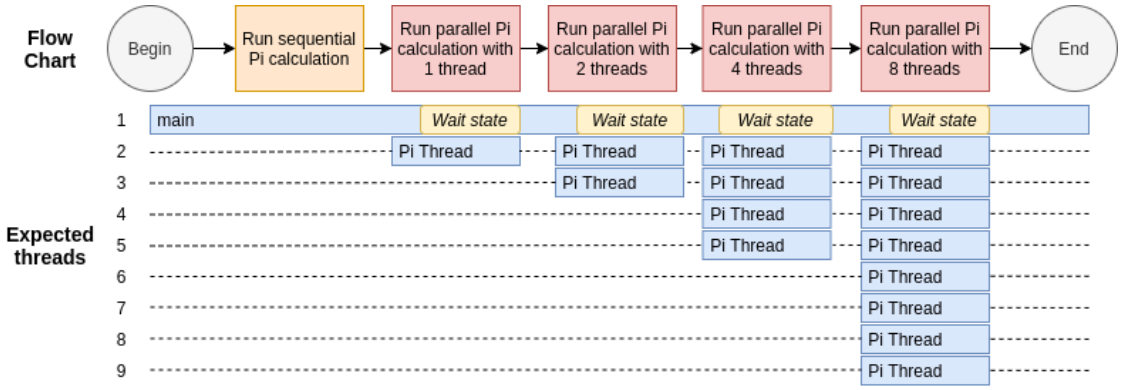

Figure 2.1: PiExample flow chart and expected threads behaviour

The `PiThreaded` class is responsible to create and run the threads:

Code 2.2: PiThreaded.java

```
public class PiThreaded
{
  long m_n;
  double m_h;
  Vector<PiThread> m_threads;

  public PiThreaded (long n, int nthreads)
  {
    m_n = n;
```

```
10      m_h = 1.0 / (double) n;
11
12      m_threads = new Vector<PiThread>(nthreads);
13      for (long i = 0; i < nthreads; i++)
14      {
15        m_threads.addElement (
16          new PiThread (m_h,
17            (n/nthreads)*i,
18            (n/nthreads)*(i+1)-1)
19        );
20      }
21    }
22
23    public void calculate()
24    {
25      /* Let the threads run */
26      for (int i = 0; i < m_threads.size(); i++)
27        (m_threads.get(i)).start();
28
29      /* Wait for their work */
30      for (int i = 0; i < m_threads.size(); i++)
31      {
32        try { (m_threads.get(i)).join(); }
33        catch (InterruptedException ignore) { }
34      }
35    }
36
37    public double result()
38    {
39      double res = 0.0;
40      for (int i = 0; i < m_threads.size(); i++)
41      {
42        /* reduce the value to result */
43        res += (m_threads.get(i)).result();
44      }
45
46      return res;
47    }
48  }
```

The classes `PiThread` and `PiSerial`, here omitted for brevity, are both responsible for the actual calculation, but the former extends Java's `Thread` class. The whole code is available in the Appendix B.

2.2 Generate the traces

Generating the traces for a Java program is done using a launcher script installed by Extrae, named `extraej`. Assuming to use the Extrae XML file contained in the examples, and assuming to have the Java class to test (named `PiExample` in this case) in the Class Path[2], the following command will launch the program and generate the traces:

```
EXTRAE_CONFIG_FILE=extrae.xml extraej -- PiExample
```

This command will generate a file called `PiExample.prv` (the name is specified in the `extrae.xml` file). By analyzing the file using Paraver, the result is shown in Figure 2.2. As it can be seen, there are not much states shown, but there are several threads detected and traced. These threads look too many compared to the ones expected (Figure 2.1). Such apparent inconsistency will be analyzed in section 2.4.

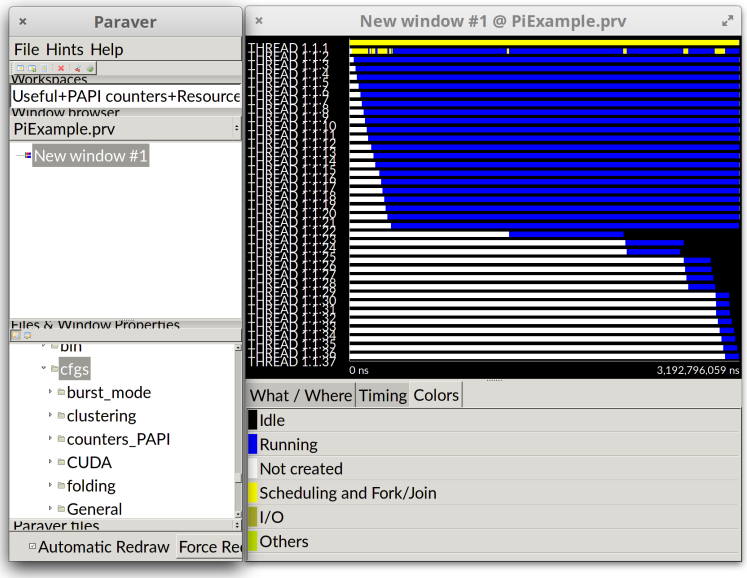

Figure 2.2: PiExample resulting traces. At the bottom a legend for the colors is reported

[2]The class path is the path where the applications, including the JDK tools, look for user classes. The default value of the class path is " . " (dot), meaning that only the current directory is searched. Specifying either the `CLASSPATH` variable or the `-cp` command line switch overrides this value [21]. By default, `extraej` does not override it, and so the compiled `.class` file must be in the current directory, or the `CLASSPATH` variable should be set accordingly.

As described in the Introduction, the most common way to instrument an application using Extrae is by using the `LD_PRELOAD` environment variable. This mechanism can be used for any program running "directly" on Linux. For this reason, even if it does not directly affect the Java program, it can be used to instrument the Java Virtual Machine (JVM).

Having a look at the `extraej` script, focusing on the part responsible to launch the program with instrumentation (Code 2.3), it can be seen how it actually uses the `LD_PRELOAD` method. An explanation of `extraej` code can be found in section 2.7.

Code 2.3: Extract of extraej showing the launching command

```
191     LD_PRELOAD=${preload} \
192     CLASSPATH=${cp} \
193       ${JAVA} ${@}
```

However, Extrae does not provide any direct instrumentation of the JVM. So how can it extract such data, without any ad-hoc instrumentation probes? The answer can be found in a mechanism that is not strictly related to Java, but to `C` and Linux: pthreads.

2.3 Pthread instrumentation

Pthread is one of the libraries instrumented by Extrae. The probes injection to trace the states of the Linux threads is done by implementing the wrappers of the valuable pthread functions like `pthread_create` or `pthread_join`.

The Java Virtual Machine (JVM) is mostly implemented in C, and for its Linux implementation each Java Thread is mapped on a POSIX thread (pthread) [22]. It is for this reason that Extrae's pthread instrumentation is effective with Java programs too.

However, this instrumentation is not enough to show all the details of a Java execution. Based on this result, it looks like that out of the pthread creation and termination no other pthread calls are employed in the JVM. As an example, it can be seen how in Figure 2.2 the synchronization operations present in the Java implementation (Code 2.2, the calls to `.join()`) are not traced[3].

[3]On top of that, I tried debugging by inserting breakpoints to other pthread calls which were never called. However, this is information is not important. As it will be discussed in later chapters, the tracing activity does not need pthread instrumentation in any case.

To let Extrae use pthread instrumentation, it must be enabled in the `extrae.xml` file as in the following piece of code (the full XML file can be found in **??**).

Code 2.4: Extract of the extrae.xml config file

```
13  <pthread enabled="yes">
14    <locks enabled="no" />
15    <counters enabled="yes" />
16  </pthread>
```

2.4 Traces analysis

By looking at the traces, several threads can be seen running throughout the process. The reported events include just "running", "join" and "I/O" events. Without knowing anything about the code and with such poor data gathered by the performance analysis, it would have been almost impossible to understand what each thread is doing. Recalling what the program does, it seems that the number of threads should be lower than the ones appearing on the traces. Indeed, the total should be $1 + 1 + 2 + 4 + 8 = 16$ threads, but on the traces the number of threads is 37.

To understand what is going on, one could try to execute the program with a different number of threads, to see how this additional number of threads behaves. By trying to instrument the sequential single-threaded program, the resulting traces are the one in Figure 2.3.

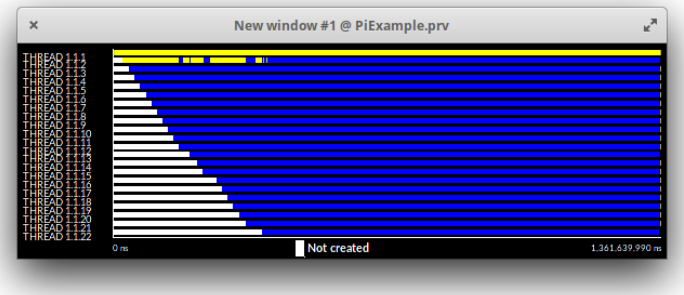

Figure 2.3: Trace of serial PiExample execution

The first thread looks exactly the same, while in the second one there are some differences related to the absence of the other threads (the yellow chunks are missing). The threads numbered from 3 to 22, instead, look the same in both of the

tracefiles. A guess[4] that can be made is that such threads are inherently launched by the JVM for each execution (Figure 2.4).

Figure 2.4: PiExample trace explained

As we will see in the following chapters, this guess will appear to be mostly correct.

The states shown in the traces (as it is shown in Figure 2.4) are reported in Table 2.1.

State	Color	State description
Idle	black	The thread stop existing
Running	blue	The thread is running, that is: it was created and it has not been destroyed yet
Not created	white	The time before the thread was created
Scheduling and Fork/Join	yellow	The thread is launching new threads
Input/Output	dark yellow	The traces are being flushed on disk
Others	light green	Other traced events, not covered by the above states

Table 2.1: Currently traced states for Java threads instrumentation

These states are valuable, but they're not enough to understand the behaviour of an application. For example, if we didn't know that, we could not say that the main thread was waiting for all the other threads before going forward. It can be deduced of course, but the state of "waiting" is not traced, so that is one event missing. All the valuable events that can be traced will be discussed in chapter 4.

[4]The methodology behind performance analysis expects some level of "guessing" when looking at the traces.

2.5 Extrae Java API through JNI implementations

When installing Extrae with Java enabled, this will install a basic instrumentation library based on JNI bindings[5]. The purpose of this library is to inject probes manually, by putting events in certain positions of the code during the development phase. This kind of events, however, do not provide any way to signal new threads, but rather they are used as "markers" for specific portions of code, relying on other mechanisms to discover on which thread they're being called—like the pthreads identifiers, in relation to the pthread instrumentation seen earlier.

The JNI implemented library is like a normal class, but with native methods instead of the standard ones. The implementation of these method is made in C language, which has to use specific functions signatures in order to be recognized by Java. In Code 2.5 and Code 2.6 can be seen an extract of the JNI bindings implemented in Extrae.

Code 2.5: Native methods inside the Java class. The instruction `System.loadlLibrary("javatrace");` dynamically loads the JNI bindings from a shared library

```
package es.bsc.cepbatools.extrae;

public final class Wrapper
{
  static { System.loadLibrary("javatrace"); }

  ...
  public static native void Event (int type, long value);
  public static native void Eventandcounters (int type, long value);
  ...
}
```

Code 2.6: C implementation of the "javatrace" library, that will be associated to the Java native methods. Notice the name of the function, that must report the package, class and method name separated by a _.

```
...

JNIEXPORT void JNICALL
Java_es_bsc_cepbatools_extrae_Wrapper_Event (JNIEnv *env,
```

[5]JNI stands for *Java Native Interface*. The JNI is a native programming interface. It allows Java code that runs inside a JVM to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. [23].

```
  jclass jc, jint id, jlong val)
{
    Extrae_event ((extrae_type_t)id, (extrae_value_t)val);
}

JNIEXPORT void JNICALL
Java_es_bsc_cepbatools_extrae_Wrapper_Eventandcounters (
  JNIEnv *env, jclass jc, jint id, jlong val)
{
    Extrae_eventandcounters ((extrae_type_t)id, (extrae_value_t)val);
}

...
```

In brief, thanks to JNI, this implementation allows to call the Extrae API for custom events instrumentation—which is written in C—directly from Java code.

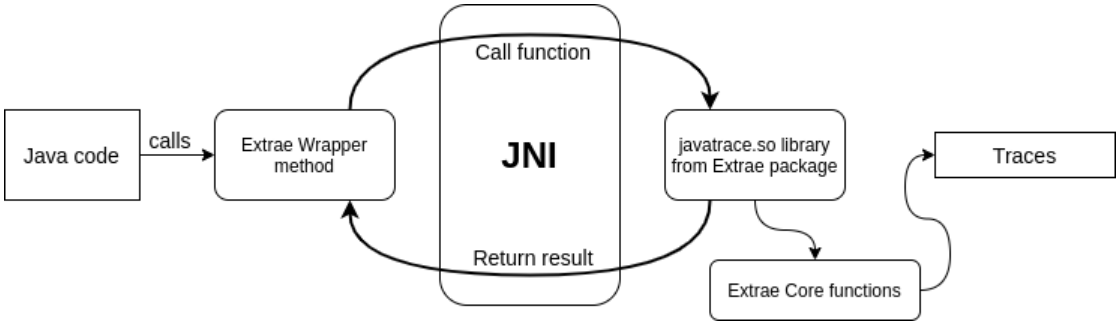

Figure 2.5: Explanation of JNI implemented Extrae wrappers

2.6 Experimental features

Besides the features presented in this chapter, Extrae's reference presented a couple described as "experimental". Such features were implemented by BSC performance tools collaborators, who don't work at the BSC anymore. For this reason, some of them were never fully implemented, and some others were not working due to some bugs accumulated in years of lack of maintenance.

All of them are explained here in this section.

2.6.1 Java Virtual Machine Tool Interface

The first one is a tracing platform based on the the JVM Tool Interface (JVM TI), a native programming interface thought for tools development. It provides both a way to inspect the state and to control the execution of applications running in the

JVM. JVM TI supports the full breadth of tools that need access to JVM state, including but not limited to: profiling, debugging, monitoring, thread analysis, and coverage analysis tools. It is available as an API for C/C++, and once the library is implemented, it can be used as an agent for the JVM [24].

Comes naturally to think that it would be a useful tool to be used to gather data for the traces—and it actually is. The idea was to implement it in Extrae, and inject it as an agent when executing the program with `extraej`.

However, despite being present among the features, it was never fully implemented. It was thought to be used as a threads-tracing mechanism instead of pthread, but its only use ended to be tracing the garbage collection events. The problem in using for threads too was about identification, because the JVM does not provide the JVM TI with a unique ID for the threads, which is instead required by Extrae's architecture. Indeed, this one was one of the issues addressed (and solved) in this thesis.

2.6.2 AspectJ for User Functions

The second experimental feature sees AspectJ, an Aspect Oriented Programming[6] extension for Java, as the tool to generate the events for the user functions. In this context, the "user functions" are those functions that the user would like to see on the traces. It should basically do what the JNI bindings were designed for, but in an automatic way by wrapping the target functions with some events, traced by Extrae—and so, placing custom events without modifying the code. Indeed, wrapping the methods with some code is basically the main purpose of AspectJ—and more in general of the Aspect Oriented Programming paradigm. The wrapper code calls the JNI implemented Extrae API library introduced in the previous section to trace the user functions events. The target user functions are listed in a file, whose path is passed to Extrae through the XML configuration file (i.e. `extrae.xml`).

This feature was also not working, but this time due to some bugs in the installation process. Once fixed, it was possible to use the "User functions" configuration of Paraver to look at them (Figure 2.6).

[6] An introduction to AOP can be found in the next chapter.

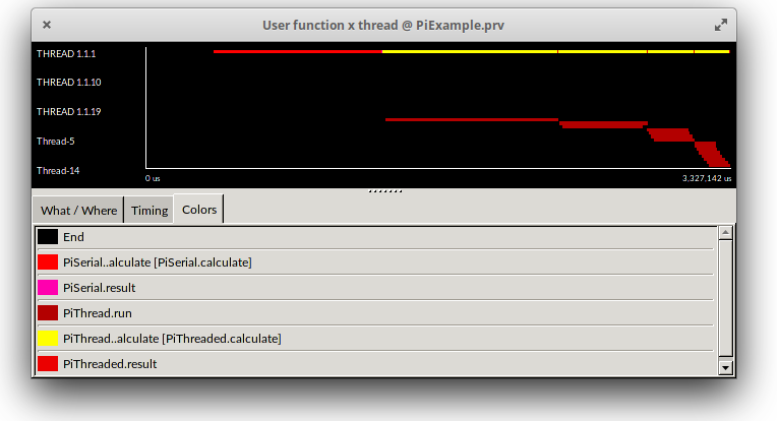

Figure 2.6: Trace showing the user functions for the PiExample program

2.7 Meet extraej

Finally, since it will be modified throughout the thesis, the state of the art version of `extraej` is worth a mention.

As said previously, `extraej` is a bash script made to launch Java applications with instrumentation. It gets some simple optional parameters:

- `-v` makes execution verbose;
- `-keep` saves the temporary files for a future use;
- `-reuse <dir>` to reuse previously instrumentation files (kept using `-keep`).

The structure of the script is quite simple. It works with some environment variables, that point to the different Extrae libraries. The main part makes all the checks for a safe execution, parses parameters and XML config file, generates the aspects for the user functions (if specified in the config file) and finally executes the Java application. The main part of `extraej` (lightened of the trivial checks) is reported in Code 2.7.

Code 2.7: Main part of `extraej`

```
#!/bin/bash
...
# Parse extraej parameters
parse_parameters "${@}"

# Do we support AspectJ?
if [[ -x "${AJC}" ]]; then
  # Do we have to reuse a previous existing instrumented package?
```

```
  if [[ "${reuse_dir}" = "" ]]; then
      # parse config file
    parse_xml ${EXTRAE_CONFIG_FILE}

        # do we have user functions to instrument?
    if [[ ${#MemberArray[@]} -gt 0 ]]; then
      tmpdir=`mktemp -d extraej.XXXXXX`

      generate_aspects

        # compile the generated aspects
      CLASSPATH=${ASPECTWEAVER_JAR}:${EXTRAEJ_JAVATRACE_PATH}:${
   CLASSPATH} \
      ${AJC} \
       -inpath . \
       -sourceroots ${tmpdir}/aspects \
       -d ${tmpdir}/instrumented
      if [[ "${?}" -ne 0 ]]; then
        die "Error! ${AJC} failed"
      fi

      execute_java ${tmpdir}/instrumented:${ASPECTWEAVER_JAR}:${
   CLASSPATH} \
        ${EXTRAEJ_LIBPTTRACE_PATH} \
        "$@"
    else
        # run without user functions instrumentation
      execute_java "${CLASSPATH}" \
        ${EXTRAEJ_LIBPTTRACE_PATH} \
        "$@"
    fi
  else
    # We want to reuse a previously instrumented app
    # Let's execute the code with Extrae support
    execute_java ${reuse_dir}/instrumented:${ASPECTWEAVER_JAR}:${
   CLASSPATH} \
      ${EXTRAEJ_LIBPTTRACE_PATH} \
      "$@"
  fi
else
  # We don't support AJC. Let's execute the code with Extrae support
  execute_java "${CLASSPATH}" \
    ${EXTRAEJ_LIBPTTRACE_PATH} \
    "$@"
fi
...
```

And `execute_java` is implemented as follows (Code 2.8). It receives the class path

and the preload as arguments, it checks for the JVM TI library availability and in case sets it as the agent.

Code 2.8: Implementation of `execute_java` procedure

```
execute_java () {
  local cp=${EXTRAEJ_JAVATRACE_PATH}:$1 # Class path
  shift
  local preload=$1    # Preload library (LIBPTTRACE)
  shift

  # Check whether Extrae supported JVMTI
  if [[ ! -r ${EXTRAEJ_LIBEXTRAEJVMTIAGENT_PATH} ]]; then
    LD_LIBRARY_PATH=`dirname ${EXTRAEJ_LIBEXTRAEJVMTIAGENT_PATH}`:${
   LD_LIBRARY_PATH} \
    LD_PRELOAD=${preload} \
    CLASSPATH=${cp} \
      ${JAVA} ${@}
  else
    LD_LIBRARY_PATH=`dirname ${EXTRAEJ_LIBEXTRAEJVMTIAGENT_PATH}`:${
   LD_LIBRARY_PATH} \
    LD_PRELOAD=${preload} \
    CLASSPATH=${cp} \
      ${JAVA} -agentpath:${EXTRAEJ_LIBEXTRAEJVMTIAGENT_PATH} ${@}
  fi
}
```

2.8 Where to go from here

In this overview there are some specific issues that emerged.

The first issue to be addressed is the lack of specific events related to Java. The only ones traced by Extrae totally rely on the pthreads instrumentation. Moreover, as pointed out in section 2.4, the traced events and states are not detailed enough to give a proper idea of the behaviour of an application.

The second one is the presence of some bugs on the already present Java instrumentation, that make Extrae not working properly with Java programs. Mainly, these bugs are related to installation issues, which used to make unavailable some features when trying to use them. Although this process has been carried out in parallel to the rest of the work, the bug-fixing operations will be omitted in the thesis discussion—unless they are particularly related to the objective, or some interesting cause and solution were found.

The final one is the absence of a mechanism to trace distributed applications. Since it is probably the most common way of executing Java on HPC, it would be

interesting to inspect this field and try to find a solution for this problem.

These issues will be the basis of the work developed next. Although not all of them have been studied throughout the thesis, they have been all kept in mind when discussing the different approaches and developing the solutions to the various problems.

Chapter 3

Java Tracing Methodologies

The main focus of this thesis is on Extrae, because the main issue is on extracting the application performance details from Java program, and not on how to visualize them. Indeed, as it has been said in the Introduction, Paraver is quite flexible and does not need any change in the code to effectively depict the events for a new instrumented framework.

Since Extrae is implemented in `C`, generating probes and wrappers would not be an issue for other C-implemented programs. Unfortunately, generating traces for a Java program cannot be so straight forward, but there are some approaches that could be tried out to extract the data needed to generate an effective trace. In this chapter there is an overview of the approaches studied in this thesis.

NB: all the discussed approaches, in order to trace the events, need to interface with Extrae's functions at some point. Such events must be coded and globally identified through the use of some constants. However, besides this small clarification, all the implementation details of such approaches are left for the next chapters.

3.1 Linker Preload approach

Once got used to Extrae, the first approach that comes to mind is the one of instrumenting the JVM using the linker preload. This kind of instrumentation expects to find the valuable functions inside the JVM, in order to define some wrappers for them, with the purpose of injecting the probes responsible for the tracing activity.

Although valuable for many frameworks, in this case this approach has not been analyzed at all. JVM internals are not standardized, and so they are not defined and immune to changes. There is no way to know that some function used in one version will be maintained in successive releases. For this reason, despite being worth a mention because of its relatedness with Extrae's standard approach, it will not be discussed further—except for possible comparison purposes.

3.2 Event-driven instrumentation

A more convenient way to trace the JVM states would be by catching the JVM events. This kind of work can be done thanks to the interface provided by the Java language: the JVM Tool Interface (JVM TI).

This approach expects to use an event-driven platform, in which the events launched by the JVM are catched by some functions containing the tracing instructions. The JVM TI allows to do that by setting user-defined callbacks, able to catch several JVM events—like thread start and end, method entry and exit, garbage collection, object allocation, etc. [25]

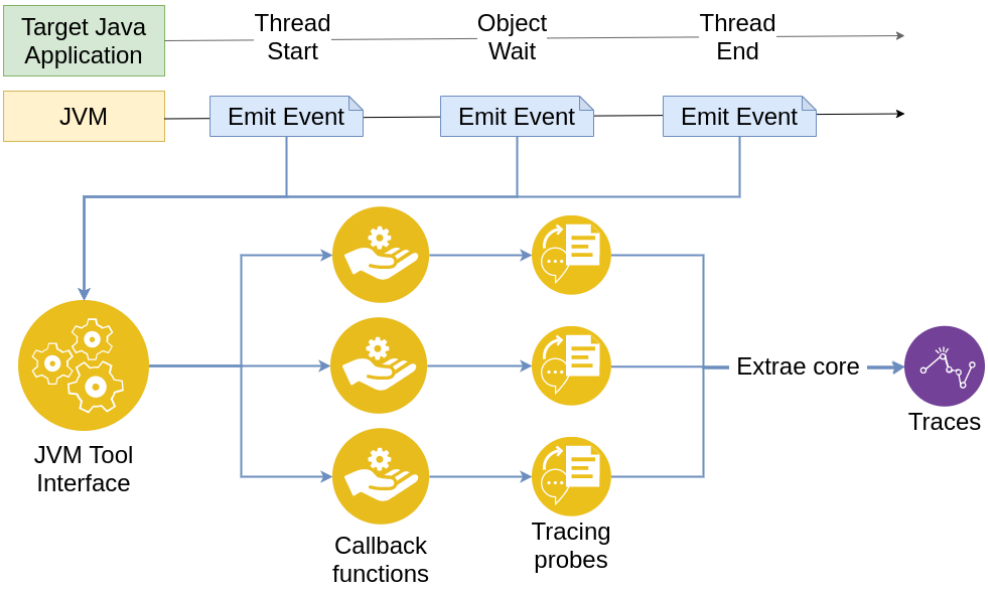

Figure 3.1: Visual explanation of event-driven approach

JVM events are essential for profiling programs, but basing the whole instrumentation on them would be somehow limiting. There are many different events catchable, but there may be interesting data depending on specific functions that do not raise any event. For example, many frameworks use busy-waiting loops implemented

in custom functions to wait for something, instead of the native `Object.wait()` method—that raises an event.

The JVM TI provides two events that may be useful in this case: method entry and method exit events. Such events are raised for each method executed by the JVM, respectively at entry and exit point. To trace some interesting methods it would be enough to monitor all these events, waiting for the interesting ones to trace the related events. Using these events is highly discouraged by the JVM TI reference guide, because of its large impact on performances. Speaking of traces in general, if the retrieved information is valuable, the great overhead generated by a callback function for each method can be acceptable. However, the problem of instrumenting specific functions can be solved in other ways. These methods are explained in the following sections.

3.3 Bytecode and Native Instrumentation

Another approach is by instrumenting the specific Java methods, using a technique called "bytecode instrumentation".

Bytecode instrumentation is a way of injecting custom instructions inside other classes, without directly modifying the code. The JVM provides some events and control functions to transform the bytecode of classes and methods. The JVM—if enabled to do it—fires an event when a class or a method are loaded. In both cases, it provides the possibility to catch the event and to read the loaded bytecode, in order to modify it[1] and re-load it again. This possibility is given through the Instrumentation API, available for both the Java language, in the package `java.lang.instrument`, and for the JVM TI, through a specific set of events and functions.

Since the injected instructions are in form of bytecodes, the instrumentation probes need to be in a form of callable Java methods. In the case of Extrae, since it is written in `C` language, such methods must rely on JNI implementations of the probes, either directly (the injected bytecodes call a JNI implemented method) or indirectly (the injected bytecodes call a Java method, which in the end calls calls one or more JNI implemented methods). The JNI implementations need to be written in `C` language, and these must rely on the Extrae's functions to trace the events.

[1] According to JVM TI reference guide [25], the modifications inserted through bytecode instrumentation can only be purely additive. Moreover, it is not possible to add new methods inside a class o to modify the signature of the other methods.

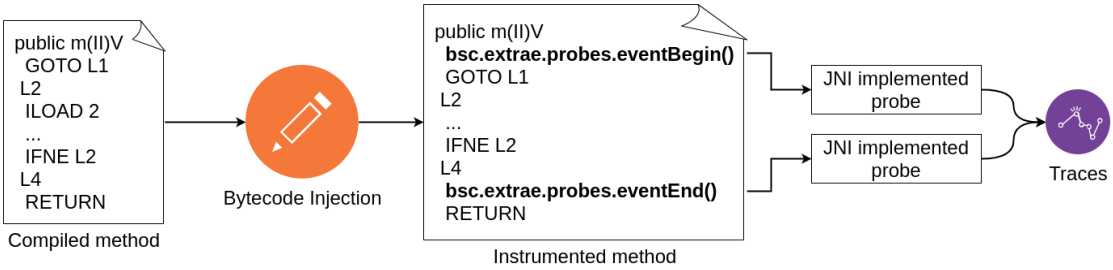

Figure 3.2: Visual explanation of the bytecode instrumentation approach

3.3.1 Bytecode manipulation in C and Java

Injecting bytecode is conceptually easy to understand, but the implementation can be tricky. The JVM provides all the tools to catch when the compiled binary is loaded and to retrieve the related bytecodes. However, these bytecodes are returned in a form of nothing more than long arrays of bytes. Injecting the bytecodes in this binary, requires essentially an intelligent array manipulation work. There are several frameworks implemented in Java for bytecode manipulation[2], but none is available for `C` language[3].

A nontrivial problem is represented by exceptions handling. For its nature, bytecode instrumentation is simply an addition in given program places. For this reason, if the program raises an exception before reaching the end of the method, it would never execute the bytecodes responsible to trace the end of the event. A possible solution would be to catch the event—available for the JVMTI—of the Exception raise, by using the agent, and to develop a mechanism to trace the end of the currently traced method.

3.3.2 Native methods instrumentation

Native methods are those methods available to be called in Java, but implemented natively in `C` language. As for the probes discussed in the previous section, this can be done thanks to the JNI. Having a C-implementation, these methods are not compiled in bytecodes, and so are not suitable to be instrumented by injecting bytecodes. This problem is solved by the Instrumentation API, both for Java and the JVM TI, which provides a way to change the name of natively implemented

[2]Among the most popular, there are ASM and Javassist.

[3]To be fair, I could find one for `C++`, but since Extrae is written in C it resulted to be quite complex to cross compile it effectively (I tried, but I had many problems at running time with the linked functions).

methods, by adding a prefix, in order to define a wrapper method (in Java).

In addition, native methods instrumentation, together with other problems, can be solved thanks to the approach explained in the next section: AspectJ.

3.4 Aspect Oriented Programming approach

Aspect Oriented programming (AOP) is a programming paradigm first introduced by G. Kiczales *et al* [26], that aims to add additional behavior to existing code without modifying the code itself. AspectJ, as one of the most popular aspect-oriented frameworks, has been chosen for this thesis.

The way of using it for tracing purposes is similar to the bytecode instrumentation approach, with the only difference in the practical implementation. Instead of using JVM events on loaded classes and inject custom bytecodes, with AspectJ it is possible to define a "pointcut" for each interesting method, and then defining the behaviour of the application before and/or after the execution [27]. This behaviour is implemented as simple Java code and, as it was for the bytecode instrumentation, it would call the JNI methods to trace the specific events.

For tracing purposes AspectJ looks very powerful, because it combines an easy Java implementation of the probes (without bytecode manipulation) and the solution to many problems given by the previous approach, like native methods instrumentation or exception handling.

3.5 Discussion on the methodology to adopt

Before trying there are no definitive conclusions to make. However, there are some approaches more promising than others. The solution proposed in this thesis will not rely on one single approach, but rather a combination of them.

Considering the needs of tracing threads behaviour and specific methods (either native or standard Java methods), the following configurations are viable solutions.

JVM TI based tracing The JVM TI is responsible to trace the events fired by the JVM, but also for the bytecode and native methods instrumentation at each class load time. Exception handling must be managed to keep coherence among the states, and bytecode needs to be manipulated in native language.

AspectJ based tracing The tracing software would rely totally on AspectJ and JNI implemented probes. Being based on methods and not JVM events, it needs

some care when defining when a new thread is created, since it could happen in different ways.

JVM TI with the aid of Java instrumentation API The JVM TI is responsible to catch and trace the JVM events, while the Java instrumentation API is responsible to trace the specific methods. Exception handling must be managed to keep coherence among the states, and the native methods should be instrumented using the Java instrumentation API. In the end, JNI methods are needed for the natively implemented probes.

JVM TI with the aid of AspectJ As the previous case, but employing AspectJ to trace the Java and native methods. Moreover, exception handling can be omitted, since adding behaviour at the end of a method can be done even when an exception is raised, without extra development effort. The drawback, with respect to the previous method, would be the required dependency of AspectJ, that needs to be installed and supported by the OS on which Extrae is going to install.

All of the above solutions require the tracing platform to be compiled as a shared library, in the case of `C/C++`, or a Java archive (JAR), in case of Java and AspectJ. In any case, they need to run as a Java agent during the target application execution.

This thesis considers the JVM TI events as tools of undoubted value. For this reason, the solution studied in the following chapters is based on the last configuration. The JVM TI will be employed for basic events tracing. For methods tracing (native methods included) AspectJ was preferred to bytecode instrumentation because of its flexibility and ease of use (and maintainability)[4].

[4]As a side note, it would be fair to say that the first solution based totally on JVM TI would be much appreciated, if Extrae was implemented in `C++`. Being implemented in `C`, it makes bytecode manipulation hard to manage. In `C++`, instead, there are some frameworks available that would have made it much simpler. I made an attempt to implement it and try to compile Extrae by mixing `C` and `C++`, but I couldn't make the linking at run-time to work properly. It would have been nice to create it externally to Extrae and compare it to the AspectJ solution, in terms of usability and generated overhead.

Chapter 4

Basic threads instrumentation with the JVM TI

4.1 JVM Tool Interface preliminaries

4.1.1 JVM TI Events

As said in the previous chapter, one of the features offered by JVM TI is the possibility of setting some callbacks for certain events. There are several available (a complete list of such events can be found on the Reference guide [25]), but the events interesting to be used for tracing purposes are the following:

- Java Thread start: generated by a new thread before its initial method executes;
- Java Thread end: generated by a terminating thread after its initial method has finished execution;
- Monitor Wait: sent when a thread is about to wait on an object;
- Monitor Waited: sent when a thread finishes waiting on an object;
- Contended Monitor Wait: sent when a thread is attempting to enter a Java programming language monitor already acquired by another thread;
- Contended Monitor Waited: sent when a thread enters a Java programming language monitor after waiting for it to be released by another thread;

- Exception: sent when an exception is raised;
- Exception catch: sent when an exception is catched;
- Garbage Collection start: sent when a garbage collection pause begins;[1]
- Garbage Collection end: sent when a garbage collection pause ends;

For each one of these events, a callback can be set.

4.1.2 JVM TI Initialization and Callbacks

Agent On Load

Implementing an agent with JVM TI requires some simple steps. First of all, `jvmti.h` needs to be included. Then, the agent must contain a function called `Agent_OnLoad`, whose structure looks as follows:

Code 4.1: JVM TI Agent On Load structure

```
1 JNIEXPORT jint JNICALL
2 Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
3 {
4     ...
5     return JNI_OK; // JNI_OK means success
6  }
```

This function is invoked when the agent library is loaded. It is used to set up all the functionalities that need to be initialized prior to the JVM.

Enabling capabilities and setting the callbacks

Any operation provided by the JVM TI can be done through an environment object, represented by a structure of type `jvmtiEnv`. To get this environment, it is necessary to call a specific function from the JVM reference—passed by the `Agent_OnLoad` function.

Code 4.2: JVM TI Environment retrieval

```
1 static jvmtiEnv *jvmti;
2
3 JNIEXPORT jint JNICALL
4 Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
5 {
```

[1] Only stop-the-world collections are reported, that are fired by the collections during which all threads cease to modify the state of the JVM.

```
6       /* Get JVMTI environment */
7       (*vm)->GetEnv(vm, (void **)&jvmti, JVMTI_VERSION);
8
9       ...
10  }
```

Before setting the callbacks for the events, it is necessary to enable the related capabilities.[2]

Code 4.3: JVM TI capabilities example

```
1  jvmtiCapabilities    capabilities;
2
3  /* Enable capability for Garbage Collection events */
4  capabilities.can_generate_garbage_collection_events = 1;
5  //set in the environment
6  (*jvmti)->AddCapabilities(jvmti, &capabilities);
```

After that, the pointer to the function can be stored into a specific data structure, which needs to be set in the JVM environment (represented by a `jvmtiEnv` structure). The signature of the callback function depends on the event and is reported in the reference guide.

Code 4.4: JVM TI event callback example

```
1  /* Setting the pointer to our callback function */
2  callbacks.GarbageCollectionStart  = &GarbageCollector_begin_callback;
3  //set in the environment
4  (*jvmti)->SetEventCallbacks(jvmti, &callbacks, sizeof(callbacks));
```

Finally, it is necessary to enable the events notifications.

Code 4.5: JVM TI notification enabling example

```
1  (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
2      JVMTI_EVENT_GARBAGE_COLLECTION_START, NULL);
```

Although the JVM TI environment can be stored in a global variable and used everywhere at any time, all these operations should be preferably executed inside the `Agent_OnLoad` function, because enabling events and callbacks are part of those operations that should be set before the JVM initialization [25].

[2] A capability is simply the ability of the JVM TI to do some operation.

Probes injection

As it can be imagined, the instructions to trace these events (i.e. the probes) will be contained in the callback functions. One event should be related to one callback, with its specific constant ID. The implementation details of such probes will be addressed in the next section.

Compiling the library and setting the agent

The JVM TI needs to be implemented and compiled as a shared library. After that, it is necessary to tell the JVM to use it as an "agent", by launching the Java program with the `-agentpath` option set to the path of the agent library.

```
java -agentpath:<path_to_jvmti_agent_library> <target>
```

The compilation will be managed by the `Makefile`[3], which build and install a shared library named `libextrae-jvmti-agent.so`. Passing the agent path as the argument will be managed by `extraej` (Code 2.3).

4.2 Tracing platform implementation

To start generating effective traces, it is necessary to define how to identify one thread, how to gather the right events and how to associate the right events to the right threads. For the Extrae core, two distinct classes of functions are going to be implemented:

- **Probe functions**, responsible to trace the specific event;
- **Backend functions**, responsible to manage the under the cover work, like notifying a new thread, giving an ID, etc.

More in general, all the Java tracing related code has been organized as follows (respecting the pre-existing Extrae source files organization):

- `tracer/wrappers/JAVA` for the above mentioned *probe* and *backend* functions;
- `java-connector/jni` for the JNI methods implementations (will depend on the previous);
- `java-connector/jvmti-agent` for the JVM TI shared library that is going to be the Java agent;

[3]Extrae uses `automake` to manage the entire build. The implementation details of the `Makefile` will not be explicitly reported.

- `java-connector/aspectj` for the Aspects implementation;
- `launcher/java` for the `extraej` launcher script;
- `merger/paraver` for the code responsible to translate the events into proper Paraver tracefiles.

Each of the previous directory will contain a `Makefile` to create the library (be it a Java archive or a shared library).

4.3 Thread identifier and Backend

4.3.1 Defining the identifier

The main problem of using the JVM TI to notify new threads is the identifier, needed by Extrae to associate the events to the right threads. When instrumenting with pthread instrumentation, that value was a pthread key[4] set by Extrae. However, by using the JVM TI to trace the different threads, the pthread instrumentation must be disabled, and with it also its thread identification mechanism. Without a defined identifier for each thread, the result would be something like Figure 4.1, in which the events are inordinately traced all on a unique thread.

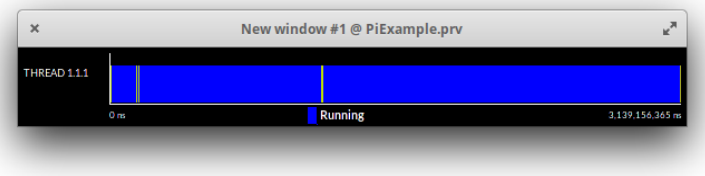

Figure 4.1: Example of traces without a defined thread identifier

Although the Java threads have an identifier, this is not available for the JVM TI[5]. Since all the Java threads are mapped on pthreads [22], even if they can't rely on the same backend functions of pthread instrumentation, they can build the

[4] `pthread_key_t` is a type of variable, initialized using the `pthread_key_create()` function and set using `pthread_setspecific()`. The create function shall create a thread-specific data key visible to all threads in the process. Key values are opaque objects used to locate thread-specific data. The same key value may be used by different threads, bu the values bound to the key by `pthread_setspecific()` are maintained on a per-thread basis and persist for the life of the calling thread. That means that it can be used to store a different ID for each a thread, retrieving it when needed using `pthread_getspecific(pthread_key)` [28].

[5] In Java, the method `Thread.getId()` returns a unique ID. However, in the thread data given by the JVM TI environment, no such ID is present.

same mechanism. Following Extrae's naming convention, the Java instrumentation backend will be implemented in a file named `java_wrapper.c`, whose header will be `java_wrapper.h`. Both the files will be contained in the proper directory, as explained previously.

4.3.2 Backend implementation

Generating the pthread identifier is done using a `pthread_key_t` global variable, that needs to be created once for all and then used by each thread to set its own ID. The creation is contextual to the initialization, so the following solution has been developed (Code 4.6).

Code 4.6: Extrae Java Backend initialization

```
#include "common.h"
#include "threadid.h"
#include "wrapper.h"
#include <pthread.h>

static pthread_key_t  pThreadIdentifier;

void Extrae_Java_Backend_init(void)
{
    // if Extrae disabled or pthread tracing enabled return
  if (! EXTRAE_ON() || Extrae_get_pthread_tracing()) return;

  // if not initialized, init Extrae
  if (! EXTRAE_INITIALIZED()) Extrae_init();

  // create pthread_key
  pthread_key_create (&pThreadIdentifier, NULL);
}
```

For each thread, the ID needs to be set when starting. Moreover, a unique ID value must be generated. A simple choice for a unique ID, is the current number of threads. This choice is not perfect, it has some pros and cons[6].

Extrae provides some backend functions to interact with the core, including the ones to define the number of threads.

[6]The consequences of this approach consist in defining the number of threads as the cumulative number, instead of the number of active threads. Doing otherwise would lead to duplicated Thread IDs (for the active ones), which can't happen for Extrae. The drawback is that defining the IDs in this way means also that the ID of a terminated thread cannot be re-utilized, making Paraver to waste much space to represent them in different lines. This approach, compared to the other solutions, has also some pros. A discussion on this is present in subsection 4.6.2.

Code 4.7: Setting the thread ID and changing the number of threads

```
1 void Extrae_Java_Backend_NotifyNewThread(void)
2 {
3     if(!EXTRAE_ON() || !EXTRAE_INITIALIZED() ||
    Extrae_get_pthread_tracing()) return;
4
5     int numthreads = Backend_getNumberOfThreads();
6     pthread_setspecific (pThreadIdentifier, (void*)((long) numthreads
    ));
7     Backend_ChangeNumberOfThreads (numthreads+1);
8 }
```

Being a multi-thread program by definition, a mutex must be added to keep the operation safe. In addition, Extrae must know the current thread ID to operate with the tracing operations. For this reason, it needs to know how to retrieve both the ID and the number of threads at any time (it lets to do so thanks to `Extrae_set_threadid_function` and `Extrae_set_numthreads_function` functions, whose parameters are pointers to functions—see Code 4.8). After some refactoring, the code looks as in Code 4.8.

Code 4.8: Refactored code to notify a new Java thread in Extrae

```
1 #include "common.h"
2 #include "threadid.h"
3 #include "wrapper.h"
4 #include <pthread.h>
5
6 static pthread_key_t  pThreadIdentifier;
7 static pthread_mutex_t  pThreadIdentifier_mtx;
8
9 void Extrae_Java_Backend_init(void)
10 {
11     if (! EXTRAE_ON() || Extrae_get_pthread_tracing()) return;
12     if (! EXTRAE_INITIALIZED()) Extrae_init();
13
14     // Tell Extrae how to retrieve current thread ID
15     Extrae_set_threadid_function(
    Extrae_Java_Backend_GetThreadIdentifier);
16     // Tell Extrae how to retrieve current number of threads
17     Extrae_set_numthreads_function(
    Extrae_Java_Backend_GetNumberOfThreads);
18     // Init identifier
19     Extrae_Java_Backend_CreateThreadIdentifier();
20 }
21
22 void Extrae_Java_Backend_CreateThreadIdentifier(void)
23 {
24     pthread_key_create (&pThreadIdentifier, NULL);
```

```
25      pthread_mutex_init (&pThreadIdentifier_mtx, NULL);
26  }
27
28  unsigned Extrae_Java_Backend_GetThreadIdentifier(void)
29  {
30      return (unsigned) ((long)pthread_getspecific(pThreadIdentifier));
31  }
32
33  unsigned Extrae_Java_Backend_GetNumberOfThreads() {
34    return (unsigned) Backend_getNumberOfThreads ();
35  }
36
37  void Extrae_Java_Backend_SetThreadIdentifier(int ID)
38  {
39      pthread_setspecific (pThreadIdentifier, (void*)((long) ID));
40  }
41
42  void Extrae_Java_Backend_NotifyNewThread(void)
43  {
44      if(!EXTRAE_ON() || !EXTRAE_INITIALIZED() ||
      Extrae_get_pthread_tracing()) return;
45
46      pthread_mutex_lock (&pThreadIdentifier_mtx);
47      int numthreads = Backend_getNumberOfThreads();
48      Extrae_Java_Backend_SetThreadIdentifier (numthreads);
49      Backend_ChangeNumberOfThreads (numthreads+1);
50      pthread_mutex_unlock (&pThreadIdentifier_mtx);
51  }
```

4.4 Notify the new threads

Once the backend is ready, it needs to be called by the JVM TI agent[7]. The JVM TI's event `JVMTI_EVENT_THREAD_START` is going to be used for this purpose. First of all a callback that calls the backend functions needs to be implemented (Code 4.9).

Code 4.9: Thread start event callback

```
#include "java_wrapper.h"

static void JNICALL Extraej_cb_Thread_start (jvmtiEnv *jvmti_env,
  JNIEnv* jni_env, jthread thread)
{
```

[7]Reminder: the file now is different. So the Java backend functions need to be included. Moreover, the library needs to be linked using automake.

```
  jvmtiThreadInfo ti;
  jvmtiError r;

    r = (*jvmti_env)->GetThreadInfo(jvmti_env, thread, &ti);
    // check if it is a valid thread
    if (r == JVMTI_ERROR_NONE){
        Extrae_Java_Backend_NotifyNewThread();
    }
}
```

Secondly, by following the same steps explained in section 4.1, the following lines need to be added to the `Agent_OnLoad` function (Code 4.10, lightened of the trivial checks).

Code 4.10: Enabling and setting the callback for the Thread start event

```
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm, char *options, void *reserved) {
    jint                rc;
    jvmtiError          r;
    jvmtiCapabilities   capabilities;
    jvmtiEventCallbacks callbacks;

    /* Init backend */
    Extrae_Java_Backend_init();

    /* Get JVMTI environment */
    rc = (*vm)->GetEnv(vm, (void **)&jvmti, JVMTI_VERSION);

    /* Get/Add JVMTI capabilities */
    memset(&capabilities, 0, sizeof(capabilities));
    // no capabilities required for Thread Start event
    r = (*jvmti)->AddCapabilities(jvmti, &capabilities);

    /* Set callbacks */
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.ThreadStart    = &Extraej_cb_Thread_start;

    r = (*jvmti)->SetEventCallbacks(jvmti, &callbacks, sizeof(
    callbacks));

    /* Enable event notification */
    r = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                JVMTI_EVENT_THREAD_START, NULL);

    return JNI_OK;
}
```

After the new threads notification mechanism is ready, by instrumenting the same example of chapter 2, the result is shown in Figure 4.2.

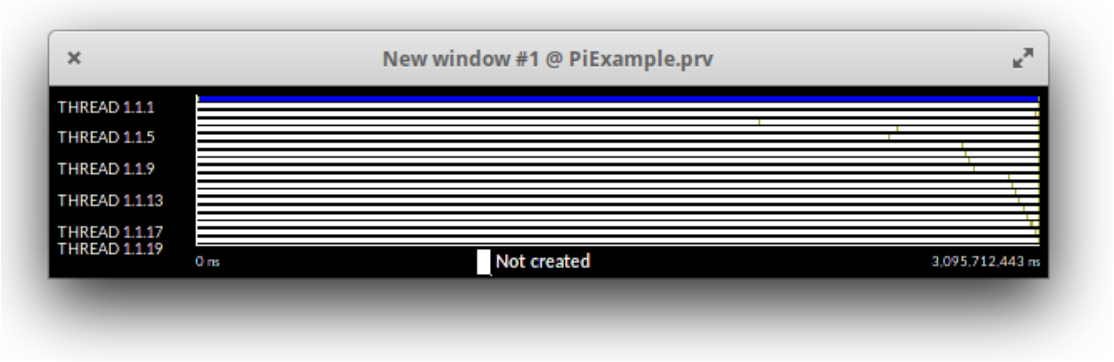

Figure 4.2: Trace with the new thread notification mechanism

As it was explained in section 2.4, the total number of threads should be 16. With pthread instrumentation it was 37, now it lowered to 19. Since no states are shown, to understand what is going on more events are necessary.

4.5 Tracing the events

4.5.1 Events IDs

First of all, an event to notify the running thread is needed. What has been done in the previous section was just a notification to Extrae on the number of threads, but it does not have any clue if the thread is running or not.

Defining and tracing new event requires two simple steps: creating a constant and unique event ID and call the Extrae internal functions to trace such event.

In addition to those, to show the events on Paraver, it is necessary to define the "semantics" of the states generated by those events and some "events description", needed to create some configuration files to show those events on Paraver in a custom way[8].

[8]Until now, the view of Paraver that has been used to show the traces was the "states" view. That means that shows the state records of the application, which represent intervals of actual thread status or resource consumption [19]. In addition, the events must have some differentiating values to let Paraver distinguish them (for example, to give them different colors). These values cannot be the events IDs alone, but they should be normalized for Paraver. This aspect will be explained later.

The chosen ID must be placed in the `events.h` file (Code 4.11).

Code 4.11: Java events IDs in `events.h` (the other IDs were pre-existent)

```
241 #define JAVA_BASE_EV                    48000000
242
243 #define JAVA_GARBAGECOLLECTOR_EV        48000001
244 #define JAVA_EXCEPTION_EV               48000002
245 #define JAVA_OBJECT_ALLOC_EV            48000003
246 #define JAVA_OBJECT_FREE_EV             48000004
247 #define JAVA_THREAD_RUN_EV              48000005 /* NEW EVENT */
```

4.5.2 Probes implementation

After having defined the ID, it needs to be traced. Such event, has a beginning and an end, so it needs to be traced twice with two different values (0 for the end and 1 for the beginning)[9]. In order to do that it is necessary to do the following:

- Add the JVM TI Thread End event callback;
- Add two new probes to trace begin and end for the event;
- Call these two probes from the Thread Start and Thread End callbacks.

Again, to follow Extrae's naming convention, these probes are implemented in a file called `java_probe.c` (Code 4.12).

Code 4.12: Probes to trace the Running Thread event in `java_probe.c`

```
1  #include "common.h"
2
3  #include "threadid.h"
4  #include "wrapper.h"
5  #include "trace_macros.h"
6
7  #include "java_probe.h"
8
9  void Extrae_Java_Probe_Thread_start(void)
10 {
11     if(! EXTRAE_ON()) return;
12
13     if (EXTRAE_INITIALIZED() && !Extrae_get_pthread_tracing())
14     {
```

[9]This choice is also attributable to the Paraver semantics. Once they're traced, Paraver normally manages the traces by looking at the "last value" of the events. So, when the event is traced with value 1 at the beginning, means that the event will remain "Up" until it's going to be set to 0 again (the end).

```
15          Backend_Enter_Instrumentation (); // Enter instrumentation
    for this thread
16
17          /* Trace JAVA_THREAD_RUN_EV with EVT_BEGIN value */
18          TRACE_MISCEVENTANDCOUNTERS(TIME, JAVA_THREAD_RUN_EV,
    EVT_BEGIN, EMPTY);
19      }
20  }
21
22  void Extrae_Java_Probe_Thread_end(void)
23  {
24      if(! EXTRAE_ON()) return;
25
26      if (EXTRAE_INITIALIZED() && !Extrae_get_pthread_tracing())
27      {
28          /* Trace JAVA_THREAD_RUN_EV with EVT_END value */
29          TRACE_MISCEVENTANDCOUNTERS(TIME, JAVA_THREAD_RUN_EV, EVT_END,
     EMPTY);
30
31          Backend_Leave_Instrumentation (); // Leave instrumentation
    for this thread
32      }
33  }
```

Notice how it checks for the pthread tracing activity, in order to avoid conflicts between the different events.

4.5.3 JVM TI Callbacks

Once implemented, these probes can be called by the JVM TI callbacks (Code 4.13). In addition to just tracing the event, Extrae gives the chance of setting the name of the thread, by calling the `Extrae_set_thread_name` function. Since the JVM TI gives the name of the thread among the info, it can be useful to retrieve this information (Code 4.13, line 16).

Code 4.13: Probes called by the JVM TI agent

```
1  #include "java_wrapper.h"
2  #include "java_probe.h"
3
4  static void JNICALL
5  Extraej_cb_Thread_start(jvmtiEnv *jvmti_env, JNIEnv* jni_env, jthread
       thread)
6  {
7    jvmtiThreadInfo ti;
8    jvmtiError r;
9
10     if (thread != NULL)
```

```
11      {
12          r = (*jvmti_env)->GetThreadInfo(jvmti_env, thread, &ti);
13
14          if (r == JVMTI_ERROR_NONE && ti.thread_group){
15              // Notify new thread
16              Extrae_Java_Backend_NotifyNewThread();
17
18              // Set the name given by Java to the thread
19              unsigned threadid = THREADID;
20              Extrae_set_thread_name(threadid, ti.name);
21
22              // Trace the start event
23              Extrae_Java_Probe_Thread_start();
24          }
25      }
26  }
27
28  static void JNICALL
29  Extraej_cb_Thread_end (jvmtiEnv *jvmti_env, JNIEnv* jni_env, jthread
        thread)
30  {
31      // Trace the end event
32      Extrae_Java_Probe_Thread_end();
33  }
```

Code 4.14: Setting the callback to the JVM TI environment

```
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
{
    ...
    callbacks.ThreadEnd = &Extraej_cb_Thread_end;
    ...
    r = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                    JVMTI_EVENT_THREAD_END, NULL);
    ...
    return JNI_OK;
}
```

4.5.4 Paraver states semantics

Finally, to show the results of these events on the trace, the semantics must be defined. Doing so is necessary to let Paraver know which state correspond to which event. There are several states in Paraver [19, p. 20], but for now the interesting ones for the thread execution are: “Idle” and “Running”.

In complex cases, the correlation between states and events may be dependent on

several factors (i.e. an event can generate different states based on the current one), but for now this definition will be straight forward: when the Thread Running event begins, the state must be "Running", when it ends it becomes "Idle".

To do so, it is enough to tell the merger[10] how to interpret the events. The java functions for the merger are contained in the file `merger/paraver/java_prv_semantics.c`, which contains a list of key-value pairs, where the key is the event ID and the value an handler function for each Java event[11]. To trace the state, it was enough to add the handler in the list and a call to the `SwitchState` function (Code 4.15). When the value of the event (`EvValue`) is different from the end, the state is Up. Otherwise, it would be set down (and other overlapping states are shown, if any).

Code 4.15: Semantics for the thread running event (`java\_prv\_semantics.c`)

```
SingleEv_Handler_t PRV_Java_Event_Handlers[] = {
  { JAVA_GARBAGECOLLECTOR_EV, JAVA_call },
  { JAVA_EXCEPTION_EV, JAVA_call },
  { JAVA_THREAD_RUN_EV, JAVA_call }, /* NEW HANDLER RECORD */
  { NULL_EV, NULL }
};

static int
JAVA_call (event_t* event, unsigned long long current_time,
  unsigned int cpu, unsigned int ptask, unsigned int task,
  unsigned int thread, FileSet_t *fset )
{
  unsigned EvType;
  unsigned long long EvValue;

  EvType  = Get_EvEvent (event);
  EvValue = Get_EvValue (event);
  switch (EvType)
  {
    case JAVA_GARBAGECOLLECTOR_EV:
    case JAVA_EXCEPTION_EV:
      Switch_State (STATE_OTHERS, (EvValue != EVT_END), ptask, task,
    thread);
      break;
    case JAVA_THREAD_RUN_EV: /* NEW CASE */
```

[10]By "merger" is simply meant the part of the application responsible to translate all the events in another format. In the specific case, it is referring to the "Paraver merger", and so it is the final part of the instrumentation, in which the events are being reported to a Paraver trace file.

[11]This list is parsed by Extrae's merger for Paraver. It simply scans all the lists for all the instrumented frameworks, calling the right handler based on the event found.

```
        Switch_State (STATE_RUNNING, (EvValue != EVT_END), ptask, task,
      thread);
        break;
    }

    trace_paraver_state (cpu, ptask, task, thread, current_time);
    trace_paraver_event (cpu, ptask, task, thread, current_time,
                  EvType, EvValue);

    return 0;
}
```

The result of all these operations can be found in Figure 4.3. As it can be seen, now the traces are showing the running state.

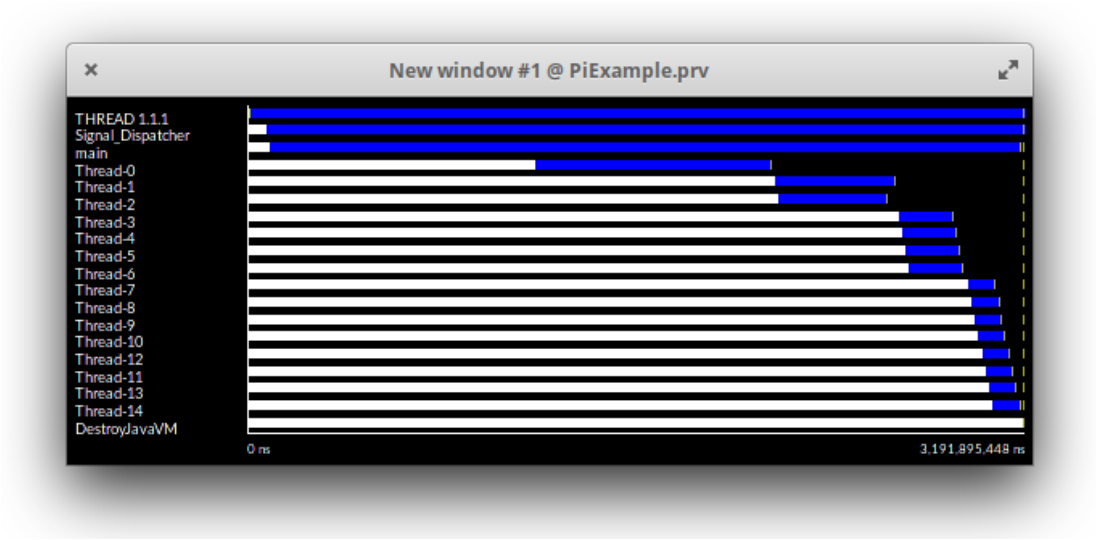

Figure 4.3: Traces with Thread Running state and thread names given by the JVM TI

4.5.5 Tracing the remaining events

The other JVM TI events need to be traced. As it has been done for the running event, all of them will be associated to a unique ID and to some state in the Paraver tracefile. Having a look to the list of the interesting JVM TI events (subsection 4.1.1), there are some missing to trace. In Table 4.1 there are all the events summarized with the related Extrae event and Paraver state.

Each of them can be traced by using a callback that calls a specific probe. The resulting traces are shown in Figure 4.4.

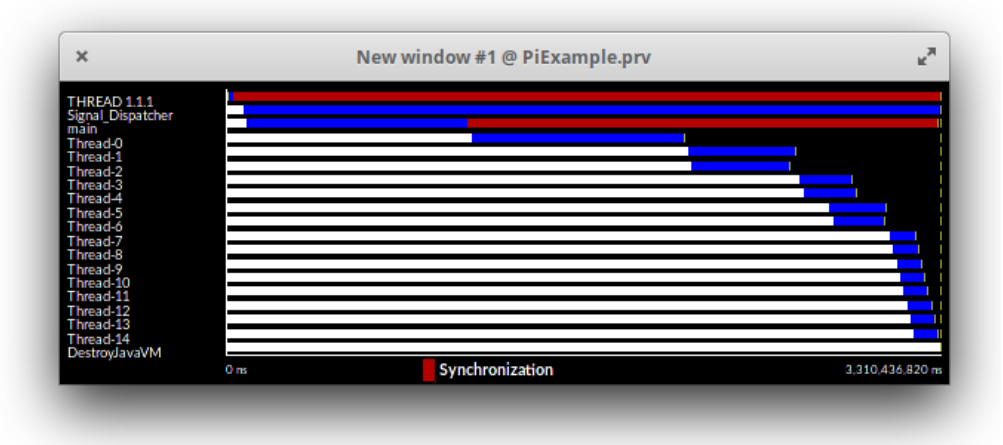

Figure 4.4: Traces with all the JVM TI events traced

JVM TI Event	Extrae Event Type	Event Value	Next State
Thread Start*	`JAVA_THREAD_RUN_EV`	Begin (1)	Running
Thread End*	`JAVA_THREAD_RUN_EV`	End (0)	Idle
Monitor Wait	`JAVA_WAIT_EV`	Begin (1)	Synchronization
Monitor Waited	`JAVA_WAIT_EV`	End (0)	*back to previous*
Contended Monitor Wait	`JAVA_WAIT_EV`	Begin (1)	Synchronization
Contended Monitor Waited	`JAVA_WAIT_EV`	End (0)	*back to previous*
Exception	`JAVA_EXCEPTION_EV`	Begin (1)	Others
Exception Catch	`JAVA_EXCEPTION_EV`	End (0)	*back to previous*
Garbage Collector start	`JAVA_GARBAGECOLLECTOR_EV`	Begin (1)	Others
Garbage Collector end	`JAVA_GARBAGECOLLECTOR_EV`	End (0)	*back to previous*

Table 4.1: Summary table for the traced Thread events and related states. The events marked with an asterisk * are disabled when pthread tracing is enabled.

4.6 Discussion of the partial results

4.6.1 Traces analysis

In this chapter the platform on which all the tracing activity will rely on has been implemented. A thread notification system has been designed and the events are starting being traced.

Moreover, in subsection 4.5.3, a name was set to the thread. Indeed, this name can be found on the left side of both Figure 4.3 and Figure 4.4. Thanks to that, it is easy to recognize what these threads are, and for some of them even their purpose

is promptly clear. Although not really informative yet, the traces are becoming to shape themselves.

4.6.2 Thread IDs

As pre-announced, defining the IDs how it is explained in this chapter leads to some consequences. The identification is done by the cumulative number of threads, which means that the threads take an increasing ID starting from 1, and the terminated threads will not see their ID re-utilized by other threads. This is the same approach applied by the pthread instrumentation.

The alternative to this approach would be a mechanism to assign IDs starting from 1, by keeping trace of the available ones and generating new IDs only if all the previous are still in use. In math symbols, every new notified thread would get as ID the smallest available m, with: $1 \leq m \leq N+1$, $N = max\{current\_thread\_IDs\}$. In other words, if at a given time the highest ID is N and a new thread is being notified, this new thread would get $N+1$ if and only if all the IDs from 1 to N are taken by active threads; otherwise, the new thread ID would be the smallest m available. This approach is the same used by the OpenMP instrumentation.

However, if for OpenMP this is the only approach that makes sense, for Java threads or pthreads it makes less sense. This assertion is based on the fact that the OpenMP threads are simple workers, while the Java threads can be specialized units with specific names, and making them share the same ID and "line" (on Paraver), would probably lead to confusions. For this reason, this solution has not been implemented in this thesis.

4.6.3 Would JVM internal instrumentation provide any added value?

As said in the previous chapter, instrumenting the JVM internals would lead to inconsistencies, soon or later. However, even ignoring this aspect, one doubt can be: would it be possible to gather more valuable data by instrumenting it anyway?

To discover some relations between the JVM events and its internal functions, I observed the execution of some Java programs using `gdb`[12]. By putting breakpoints in the JVM TI callbacks, inspecting the call stack and spending some time to analyze different programs, I discovered that most of the valuable events and

[12]GDB is the GNU Project Debugger, a debugger that can be used for many languages (`C` and `C++` included). It allows to set breakpoints, stop the program on certain conditions, inspecting local variables or the call stack and other helpful operations [29].

interesting functions correspond to some of the above events. Instrumenting such functions, would not give any additional information respect to the events and data provided by the JVM TI. The only exception is the "Object notify" event: when it happens, always the same internal function is called, but it does not raise any event. However, `Object.notify()` is a native method, and it can be instrumented in other ways.

In conclusion, the JVM TI makes it pointless to even try the manual instrumentation of the JVM internal functions. And this is also true thanks to the next tool that is going to be employed in this study, and that will be explained in the next chapter: AspectJ.

Chapter 5

AspectJ and other improvements

The JVM TI gives access to a set of very useful events, which can be effectively used to trace the threads states. However, there are still other threads missing to trace, which are not available through the JVM TI. The necessity of a more general approach raises, an approach that is not tied to the JVM events available via the Tool Interface, but one that allows to instrument freely some target methods. Such approach, can be found in the Aspect Oriented Programming, and more specifically in AspectJ.

However, before diving into the implementation of the AspectJ solution, some preliminary work is necessary. Indeed, it would not be possible to instrument complex applications before modifying `extraej`, because it currently does not support custom class paths.

Moreover, the JVM TI put at the developer's disposal a Thread internal storage, similar to the pthread key explained in the last chapter. This can be used to store important information, such as the thread ID or even more things. Some refactoring would be needed to use as the ID storage though.

Finally, recalling the discussion about the events information to depict on Paraver (subsection 4.5.4), the events should be represented in some way other than just the states records.

All these improvements will be studied throughout this chapter.

5.1 Setting user class path to `extraej`

The state of the art of `extraej` (section 2.7) needs to be launched in the same directory of the class to instrument, because the standard class path when launching a java application is "." (the current directory). This is limiting for several reasons:

- it lets to instrument only the applications that depend only on the JDK, and not on other frameworks whose classes are placed somewhere else;
- it does not let to instrument most of the applications object of this thesis, since the frameworks are not part of the JDK;
- seen the previous point, and as it will be discussed in section 5.2, the AspectJ's compiler (`ajc`) needs the classes to instrument in the class path to work.

In order to improve this, an option `-cp` was added to `extraej`, and works as follows:

```
extraej -cp <DIRECTORY_OR_FILE> -- <TARGET>
```

That is implemented in order to add to the class path all the containing files if its a directory, or just the file if not[1] (Code 5.1).

Code 5.1: `extraej` new option implementation

```
#!/bin/bash
...
parse_parameters () {
    ...
    elif [[ "${1}" = "-cp" ]]; then
      shift
            user_cp=${1}
    ...
}
...
if [[ "${user_cp}" != "" ]]; then
  if [[ -d "${user_cp}" ]]; then
    for cp in $(ls ${user_cp}); do
      CLASSPATH="${user_cp}/${cp}:${CLASSPATH}"
    done
  else
      CLASSPATH=${user_cp}:${CLASSPATH}
```

[1] The class path would be enough to point to a directory full of compiled `.class` files. However, this way is more general, because if in the directory are present JAR files (which is quite common when building the dependencies with Maven, for example), they will be added to the class path too.

```
  fi
  echo "Updated Classpath: ${CLASSPATH}"
fi
...
```

5.2 AspectJ for Instrumentation

5.2.1 Introduction to AspectJ

AspectJ is a framework to implement aspect-oriented programming Java. In practice, it adds to the Java language just one new concept: a join point. This is implemented and managed through some new constructs: pointcuts, advice, inter-type declarations and aspects.

The official "Getting started with AspectJ" guide [30] defines these new elements as follows:

"A *join point* is a well-defined point in the program flow. A *pointcut* picks out certain join points and values at those points. A piece of *advice* is code that is executed when a join point is reached."

"AspectJ's *aspect*s are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations."

Code 5.2: Example of an *aspect* implementation in AspectJ, with some local attributes and methods, a *pointcut* to define the type of call to catch and an *advice* to add some behaviour one the catched call. The aspects are normally stored in `.aj` files.

```
1  import org.aspectj.lang.reflect.*;
2  import java.lang.*;
3
4  aspect PointObserving {
5      OutputStream logStream = System.err;
6      boolean updatedX = false;
7
8      void logChangeX(int x) {
9          logStream.println("about to change X");
10     }
11
12     pointcut changeX(): call(void Point.setX(int));
13     before(): changeX() {
14         updatedX = true;
15         logChangeX();
16     }
```

```
17 }
```

This instrumentation approach is going to use AspectJ just for calling the probes, which are going to be implemented using JNI[2]. That means, the *pointcuts* will be on the methods to instrument, while the pieces of *advice* will be implemented in order to call the JNI methods[3], as shown in the example Code 5.3.

Code 5.3: Example of instrumented methods using AspectJ

```
1  import org.aspectj.lang.reflect.*;
2  import bsc.extrae.JavaProbes;
3
4  aspect SomeClassInstrumentation {
5
6      pointcut someOperation(): call(void SomeClass.someMethod(void));
7
8      before(): someOperation() {
9          JavaProbes.traceSomeEventBegin();
10     }
11
12     after(): someOperation() {
13         JavaProbes.traceSomeEventEnd();
14     }
15
16 }
```

5.2.2 What to trace using AspectJ

Before diving into the development phase, the methods to be instrumented must be decided.

The first event interesting to gather can be found by looking at the traces given by the pthread instrumentation (Figure 2.2) and the ones given by the instrumentation through JVM TI (Figure 4.4). In the second one, even if the synchronization state is present, the "Scheduling and fork/join" state is totally missing. A thread is in this state while it is creating a new thread, and in our example such thread creation

[2]The basic concepts of *Pointcut*, *Advice* and *Aspect* are enough to deliver this job. For this reason, this brief overview of AspectJ ends here. AspectJ's power is far beyond solving this simple task, but all the possible features and complex usages won't be explained, because not related to the objective of the thesis. However, any necessary tool will be gradually introduced.

[3]This is almost how it works with the User Functions, explained in section 2.5. The generation was different, but the concept of calling the JNI methods inside the pieces of advice was the same.

is what is done by the `Thread.start()` method[4]. So this is the first method to be traced using AspectJ.

Another event that may be interesting to trace, and of which there are no clue in JVM TI events, is the monitor notification event (`Object.notify()`).

These two (`Thread.start()` and `Object.notifiy()`) will be the first two instrumented methods using AspectJ.

5.2.3 JNI implemented probes

As already stated, the probes must be implemented through JNI bindings, in order to interoperate with Extrae's core in C language. In order to implement the probes using the JNI (recalling section 2.5), it is necessary to define a class with some native methods (Code 5.4).

Code 5.4: Extrae Java instrumentation probes Class

```
1  package es.bsc.cepbatools.extrae;
2
3  public final class JavaProbes
4  {
5      static { System.loadLibrary("javatrace"); }
6
7      public static native void ThreadStartBegin();
8      public static native void ThreadStartEnd();
9
10     public static native void ObjectNotifyBegin();
11     public static native void ObjectNotifyEnd();
12 }
```

As it can be seen by Code 5.4, the loaded library is "javatrace", the same of the wrappers in Code 2.5, and also the package is the same. Indeed, the JNI implemented probes are going to end in the same library, in order to keep the same dependencies and `Makefiles`. Indeed, from the implementation point of view, these probes are not different from the functions of the Java Extrae instrumentation API, presented in section 2.5.

The probes are implemented in C language as in Code 5.5, and they will contained in the same directory of the other JNI bindings (`java-connector/jni`), in a file called `extrae_javaprobes.c`.

[4]Indeed, the Java threads work in this way. In order to run the thread (that means executing what is contained in the Thread Object's `run` method), the `start` method must be called on the Thread object (it can be seen in Code 2.2 too).

Code 5.5: Extrae Java instrumentation probes implementation in C language

```
JNIEXPORT void JNICALL
Java_es_bsc_cepbatools_extrae_JavaProbes_ThreadStartBegin(
  JNIEnv *env, jclass jc)
{
    Extrae_Java_Probe_ThreadStart_begin ();
}

JNIEXPORT void JNICALL
Java_es_bsc_cepbatools_extrae_JavaProbes_ThreadStartEnd(
  JNIEnv *env, jclass jc)
{
    Extrae_Java_Probe_ThreadStart_end ();
}

JNIEXPORT void JNICALL
Java_es_bsc_cepbatools_extrae_JavaProbes_ObjectNotifyBegin(
  JNIEnv *env, jclass jc)
{
    Extrae_Java_Probe_ObjectNotify_begin ();
}

JNIEXPORT void JNICALL
Java_es_bsc_cepbatools_extrae_JavaProbes_ObjectNotifyEnd(
  JNIEnv *env, jclass jc)
{
    Extrae_Java_Probe_ObjectNotify_end ();
}
```

The probes are implemented as in subsection 4.5.2, in the same file (Code 5.6). Notice how the Thread Start event checks for the pthread tracing activity, while the Object Notify event does not. In this way, Extrae can trace the notification events even during the pthread tracing (as it was already happening for all the events except from the Thread Running one).

Code 5.6: The new probes implemented in `java_probe.c`

```
void Extrae_Java_Probe_ThreadStart_begin(void)
{
  if(!EXTRAE_ON() || !EXTRAE_INITIALIZED() ||
    Extrae_get_pthread_tracing()) return;

  TRACE_MISCEVENTANDCOUNTERS(TIME, JAVA_THREAD_START_EV, EVT_BEGIN,
                             EMPTY);
}

void Extrae_Java_Probe_ThreadStart_end(void)
{
```

```
11    if (!EXTRAE_ON() || !EXTRAE_INITIALIZED() ||
        Extrae_get_pthread_tracing()) return;
12
13    TRACE_MISCEVENTANDCOUNTERS(TIME, JAVA_THREAD_START_EV, EVT_END,
14                               EMPTY);
15  }
16
17  void Extrae_Java_Probe_ObjectNotify_begin(void)
18  {
19    if (!EXTRAE_ON() || !EXTRAE_INITIALIZED()) return;
20
21    TRACE_MISCEVENTANDCOUNTERS(TIME, JAVA_OBJECT_NOTIFY_EV, EVT_BEGIN,
22                               EMPTY);
23  }
24
25
26  void Extrae_Java_Probe_ObjectNotify_end(void)
27  {
28    if (!EXTRAE_ON() || !EXTRAE_INITIALIZED()) return;
29
30    TRACE_MISCEVENTANDCOUNTERS(TIME, JAVA_OBJECT_NOTIFY_EV, EVT_END,
31                               EMPTY);
32  }
```

After that, it is necessary to specify which states they trace. As already said, the state related to the `Thread.start()` method is of the type "Scheduling and Fork/Join", while the state related to `Object.notify()` will be one that has not been mentioned yet: "Group Communication". Once that everything is set, the only remaining step is to call the probes and let the tracing happen.

5.2.4 Instrumentation aspects implementation

Implementing the Aspect is quite straight forward. It is necessary to define two pointcuts to catch the two methods. Moreover, some care is needed in order to avoid some inconveniences. Since the class `Object` is being instrumented, and since all the classes are sub-classes of it, it must be told AspectJ that the events within the Aspect execution must not be included in the pointcut itself. The code looks like Code 5.7.

Code 5.7: `Extrae.aj`: aspect implementation to call the probes before and after the instrumented methods

```
1  package extrae.aspects;
2
3  import org.aspectj.lang.reflect.*;
4  import java.lang.*;
5
```

```
6  public aspect Extrae {
7      // !within to avoid inner objects instrumentation
8      pointcut Tread_Create(): !within(extrae.aspects.Extrae)
9                           && call (* java.lang.Thread.start (..));
10
11     pointcut Notify(): !within(extrae.aspects.Extrae)
12                          && call (* java.lang.Object.notify (..));
13
14     before() : Tread_Create()
15     {
16         es.bsc.cepbatools.extrae.JavaProbes.ThreadStartBegin();
17     }
18
19     after() returning () : Tread_Create()
20     {
21         es.bsc.cepbatools.extrae.JavaProbes.ThreadStartEnd();
22     }
23
24     before() : Notify()
25     {
26         es.bsc.cepbatools.extrae.JavaProbes.ObjectNotifyBegin();
27     }
28
29     after() returning () : Notify()
30     {
31         es.bsc.cepbatools.extrae.JavaProbes.ObjectNotifyEnd();
32     }
33 }
```

In order to be available for all the executions in Java, all the aspects are placed in a specific system directory during the installation. The chosen directory is `${EXTRAE_LIB_DIR}/extraejaspects`.

5.2.5 Compiling everything and setting the agent

The aspects can be combined to the actual Java program with an operation called "Weaving". This can be delivered in three different ways:

- Compile-time weaving, consisting in compiling the Java and AspectJ source codes together;
- Post-compile weaving (also sometimes called binary weaving), used with existing class or JAR files, that are already compiled in bytecodes;
- Load-time weaving (LTW), that expects to defer the binary weaving until a class loader loads a class file and defines the class to the JVM.

The simplest ones are the first two, while the latter requires some additional settings and precautions. The one used for the User functions (subsection 2.6.2) is the second one, and so the proposed solution will use the "Post-compile weaving" as well. The compilation with AspectJ needs to be done using AspectJ's own compiler, named `ajc`. It is a command similar to `javac`, and takes some option to define the source code (aspects file) and the input classes to instrument.

The AspectJ compiler `ajc` needs to have in the class path all the classes to weave[5]. It will then output the new compiled classes, which will be the ones executed to gather the traces. Input and output files will all be collected in a temporary directory: the aspects will be copied inside `<temp_directory>/aspects` and the instrumented classes will be placed inside `<temp_directory>/instrumented`. All these operations are summarized in Figure 5.1.

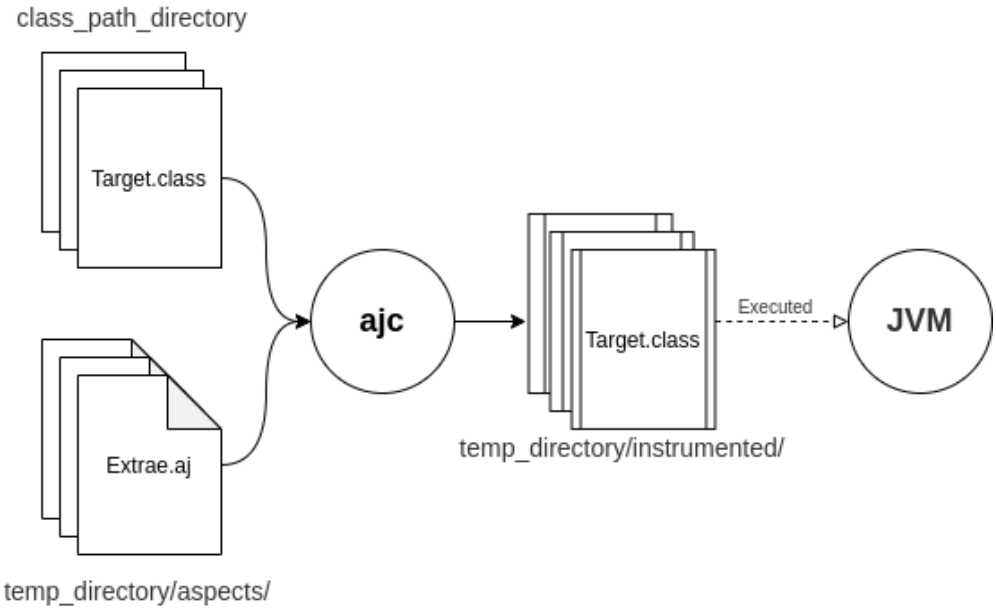

Figure 5.1: Instrumentation process using AspectJ compilation

The compilation is going to be implemented inside `extraej` (Code 5.8), just before launching the java application.

Code 5.8: Extract of `extraej`, containing the aspects compilation

```
tmpdir=‘mktemp -d extraej.XXXXXX‘
```

[5]For now, this does not represent a problem, since the instrumented classes are part of the standard Java language. However, when the classes will be some external frameworks, some care will be needed when compiling.

```
mkdir ${tmpdir}/aspects

if [[ "${user_cp}" != "" ]]; then
  inpath=${user_cp}
else
  inpath=.
fi

cp -r ${EXTRAEJ_ASPECTS_DIR}/* ${tmpdir}/aspects/

CLASSPATH=${ASPECTJRT_JAR}:${ASPECTWEAVER_JAR}:${
    EXTRAEJ_JAVATRACE_PATH}:${CLASSPATH} \
${AJC} \
  -1.8 \
  -inpath ${inpath} \
  -sourceroots ${tmpdir}/aspects \
  -d ${tmpdir}/instrumented
```

And then, launching the application will be done by adding the directory with the instrumented classes `<temp_dir>/instrumented` to the class path[6] (Code 5.9).

Code 5.9: Extract of `extraej`, containing the Java launching instruction

```
execute_java ${tmpdir}/instrumented:${ASPECTJRT_JAR}:
     ${ASPECTWEAVER_JAR}:${CLASSPATH} \
     ${EXTRAEJ_LIBPTTRACE_PATH} \
     "$@"
```

All these operations do not alter the original command of execution (the "user class path" option is not needed, because the instrumentation is on the classes `Thread` and `Object`, which are part of the JDK).

5.2.6 Resulting traces and discussion

To show the new implemented features, a new example Java program has been instrumented (Figure 5.2).

Without looking at the code, but zooming on the main part (Figure 5.3), the behaviour of the application can almost be understood with just looking at the traces. The main thread generates the other two (because no scheduling is present on the other threads), `Thread-0` waits until `Thread-1` notifies something, and then, after some operations, they both die and the program ends (also, the main thread was waiting for both without doing anything else).

[6] The first argument of `execute_java` represents the class path (section 2.7, Code 2.8).

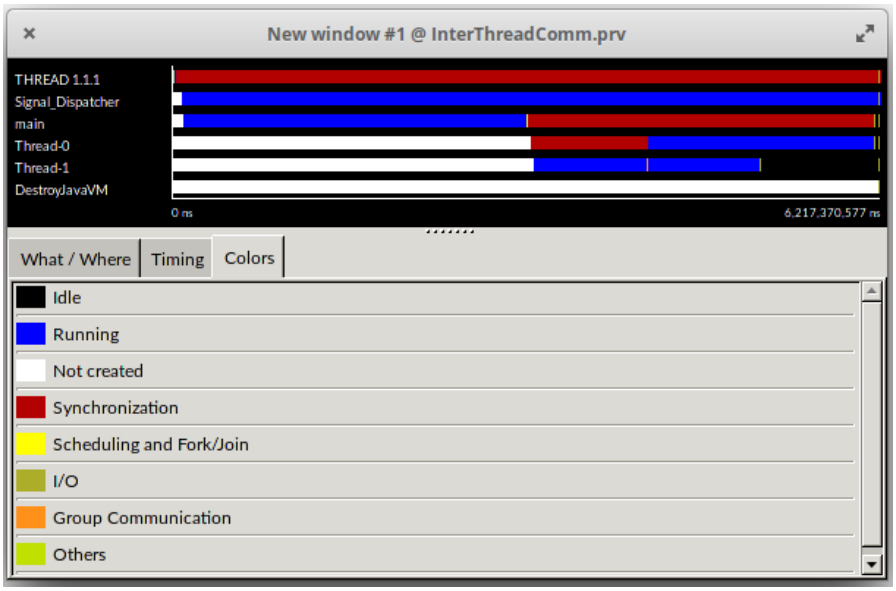

Figure 5.2: Traces of an instrumented application, with the new traced states: Scheduling and Group Communication

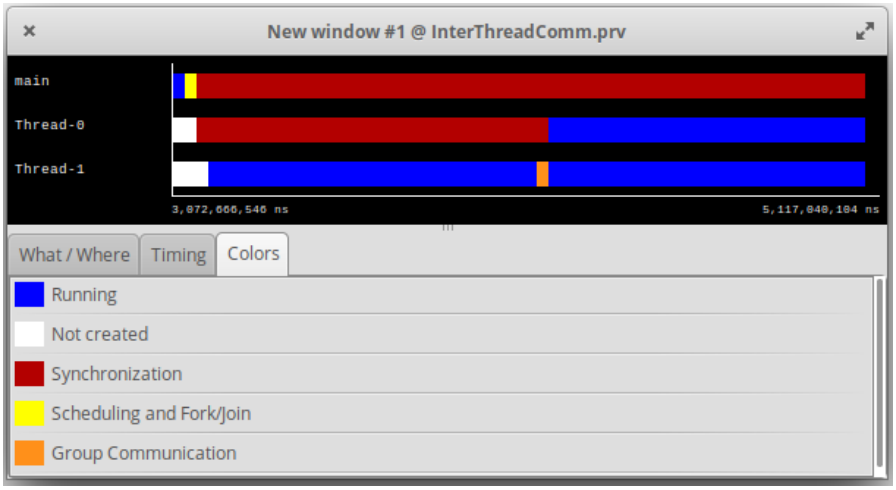

Figure 5.3: Traces of an instrumented application, focus on the main part. It can be noticed the scheduling state on the main thread, while it is generating `Thread-0` and `Thread-1`, and also the notification event on `Thread-1` that "unlocks" `Thread-0` from the waiting state.

5.3 Events values: a better view

Using Paraver to show the states was enough to understand the program behaviour, but what about the specific operations? Understanding that a "Group communication" is related to an `Object.notify()` call, or that the "Synchronization" state is related to `Object.wait()`, requires a level of knowledge of the Extrae's instrumentation package implementation. Moreover, no clue is given about the "Others" state, or if more events are related to the same state, making them the same from the point of view of the state. This may not be a problem, if just a general understanding of the program is the finale objective. However, it's inevitably a loss of information that can (and should) be fixed.

Fortunately, Paraver is able to show different events with different colors. The problem is that it bases this differentiation on the value of the event, and not the type. More specifically, all the events traced so far reported all the same values when traced: `EVT_BEGIN` and `EVT_END` (Code 4.12, Code 5.6). For this reason, Paraver would not be able to distinguish among the different events.

In order to show the Java events in different colors, an addition to the Paraver merger[7] has been made. When tracing the event, it is mapped to a different pair type-value. The mapping is shown in Table 5.1[8].

The mapping is simply implemented as an array of normal C `structs`, that is iterated if the event is of one the Java ones. In addition to the mapping, Paraver requires the configuration to be print on the `.pcf` file, in order to associate the event values to the labels. The implementations of the mapping (Code 5.10) and the printing of the labels configuration for the `.pcf` file (Code 5.11), are both in the `merger/paraver/java_prv_events.c` file.

Code 5.10: Extract of the file `java_prv_events.c`, mapping

```
1 void Translate_JAVA_MPIT2PRV (int typempit, UINT64 valuempit, int *
      typeprv, UINT64 *valueprv)
2 {
3   int index = find_event_mpit(typempit);
4
```

[7]The Paraver merger has been introduced in subsection 4.5.4, when talking about the states semantics).

[8]On the table it can be seen how the event `JAVA_THREAD_RUNNING_EV` is not being mapped to a new event type. The reason behind the choice is based on the fact that it would have made the trace more confusing then how it would have added value. In this way, the focus can be on the single operations. Understanding when a thread is generated can be done thanks to the `Thread.start()` event. In any case, Paraver allows to add it to the filter in order to show it together with the other events.

Extrae event type	New type	New value	Event label
JAVA_GARBAGE_COLLECTOR_EV	JAVA_BASE_EV	1	Garbage Collection
JAVA_EXCEPTION_EV	JAVA_BASE_EV	2	Exception
JAVA_OBJECT_ALLOC_EV	JAVA_BASE_EV	3	Object allocated
JAVA_OBJECT_FREE_EV	JAVA_BASE_EV	4	Object freed
JAVA_WAIT_EV	JAVA_BASE_EV	5	Monitor wait
JAVA_THREAD_START_EV	JAVA_BASE_EV	6	Thread scheduling
JAVA_OBJECT_NOTIFY_EV	JAVA_BASE_EV	7	Monitor notify
All the events with value 0	-	0	Outside Java events

Table 5.1: Events values mapping for Paraver. Valid for `EVT_BEGIN` only, when the original event value is `EVT_END` (that is 0) the new event value will be 0.

```
5    if (index >= 0)
6    {    // if found the mapping, change with new type and value
7      *typeprv = event_mpit2prv[index].type_prv;
8      *valueprv = (valuempit!=0)?event_mpit2prv[index].val_prv:0;
9    }
10   else
11   {    // otherwise, keep the input ones
12     *typeprv = typempit;
13     *valueprv = valuempit;
14   }
15 }
```

Code 5.11: Extract of the file `java_prv_events.c`, printing the labels

```
1  void JavaEvent_WriteEnabledOperations (FILE * fd)
2  {
3    int atleastone = FALSE;
4    for (unsigned i=0; i < MAX_JAVA_INDEX && !atleastone; i++)
5      atleastone |= event_mpit2prv[i].in_use;
6    if (!atleastone) return; // if none is used, don't print anything
       and return
7
8    fprintf (fd, "EVENT_TYPE\n");
9    fprintf (fd, "0 %d Java basic events", JAVA_BASE_EV);
10   fprintf (fd, "VALUES");
11
12   fprintf (fd, "0 Outside thread execution\n");
13   for (unsigned i=0; i < MAX_JAVA_INDEX; i++)
14   {
15     if (event_mpit2prv[i].in_use) // if used in the trace, print
       value and label
```

```
16          fprintf(fd, "%d %s\n", event_mpit2prv[i].val_prv,
        event_mpit2prv[i].label);
17      }
18  }
```

The translation function is then called by the `trace_paraver_event` function, which is the one that actually prints the numeric values on the trace (Code 5.12).

Code 5.12: Extract of the file `paraver_generator.c`, `trace_paraver_event` function

```
void trace_paraver_event (
    unsigned int cpu, unsigned int ptask, unsigned int task,
    unsigned int thread, unsigned long long time, unsigned int type,
    UINT64 value)
{
     ...
     if (type >= MPI_MIN_EV && type <= MPI_MAX_EV)
   {
     Translate_MPI_MPIT2PRV (type, value, &tipus, &valor);
   }
   else if (type >= JAVA_MIN_EV && type <= JAVA_MAX_EV) /* if a Java
     event */
   {
     Translate_JAVA_MPIT2PRV (type, value, &tipus, &valor); /* change
     to new type and value */
   }
   else
   {
     tipus = type;
     valor = value;
   }
   ...
}
```

By instrumenting using this implementation, and by changing the view of Paraver in order to filter by the `JAVA_BASE_EV`, the result for the traces are shown in Figure 5.4 (the PiExample program from chapter 2) and Figure 5.5 (the new example program of subsection 5.2.6).

Using this view, together with the state view, will give an holistic understanding of the traced events for all the Java applications. This and the other improvements contained in this chapter will give an additional support for analyzing a real world case study, which is going to be the topic of the next chapter.

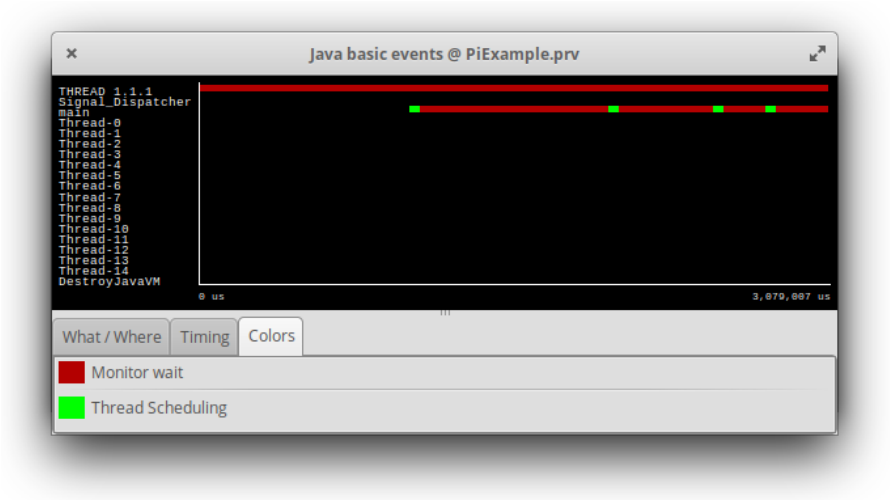

Figure 5.4: PiExample trace with the Java Paraver view

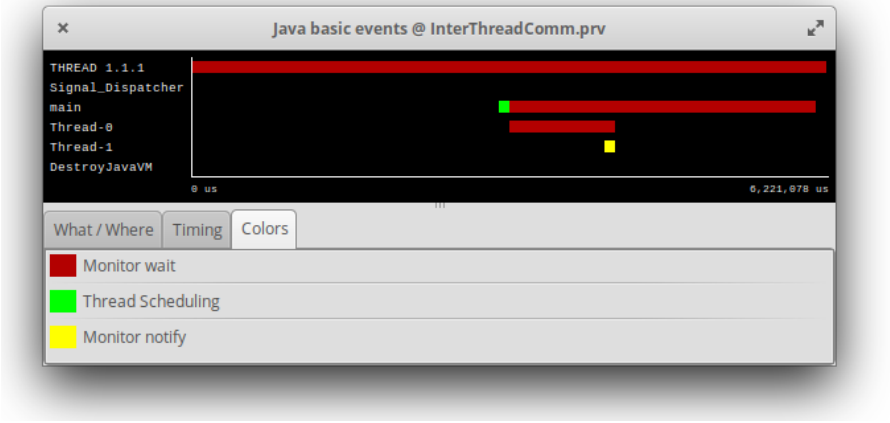

Figure 5.5: New example trace with the Java Paraver view

Chapter 6

Case Study: Hadoop MapReduce

6.1 What methods to instrument

6.2 Probes implementation

6.3 Aspects generation

6.4 Distributed execution

Chapter 7

Discussion of Results

7.1 What to Look for

7.2 Tracing overhead analysis

7.3 Further Improvements

Chapter 8

Conclusions

Appendix A

Environment set-up

Appendix B

Extrae State of the Art complete code (with the Example)

Bibliography

[1] T. Sterling, M. Anderson, M. Brodowicz. *High Performance Computing. Modern systems and practices*. Cambridge, MA, USA: Morgan Kauffman, 2018 (cit. on pp. 1, 2).

[2] A. Heck, F. Murtagh. «Artificial intelligence applications for Hubble Space Telescope operations». In: *Knowledge-Based Systems in Astronomy*. Heidelberg, Germany: Springer, 1989. Chap. 1, pp. 3–31 (cit. on p. 1).

[3] *Hubble Telescope photograph*. URL: `https://hubblesite.org/image/4679/gallery` (cit. on p. 1).

[4] Kaggle. *2019 Kaggle ML DS Survey*. Tech. rep. Kaggle, 2019 (cit. on p. 2).

[5] International Data Corporation (IDC). «Worldwide Spending on Artificial Intelligence Systems Will Grow to Nearly $35.8 Billion in 2019». In: *Worldwide Semiannual Artificial Intelligence Systems Spending Guide* (2019). URL: `https://www.idc.com/getdoc.jsp?containerId=prUS44911419` (cit. on p. 2).

[6] I.H. Witten, E. Frank, M.A. Hall, C.J. Pal. *Data Mining. Practical Machine Learning Tools and Techniques*. Fourth. Cambridge, MA, USA: Morgan Kauffman, 2017. Chap. Appendix B (cit. on p. 2).

[7] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica. «Spark: Cluster Computing with Working Sets». In: UC Berkley, California, USA, June 2011, p. 1. URL: `https://amplab.cs.berkeley.edu/wp-content/uploads/2011/06/Spark-Cluster-Computing-with-Working-Sets.pdf` (cit. on p. 3).

[8] E. Sparks, A. Talwalkar. «Spark Meetup: MLbase, Distributed Machine Learning with Spark». In: San Francisco, California, USA, June 2013. URL: `http://www.slideshare.net/chaochen5496/mlllib-sparkmeetup8613finalreduced` (cit. on p. 3).

[9] G. Ingersoll. «Introducing Apache Mahout». In: *IBM developers Archives* (2009). URL: `https://www.ibm.com/developerworks/java/library/j-mahout/` (cit. on p. 3).

[10] *MapReduce*. Wikipedia.org, 2020. URL: https://en.wikipedia.org/wiki/MapReduce (cit. on p. 3).

[11] A. Koliopoulos, P. Yiapanis. F. Tekiner, G. Nenadic, J. Keane. «A Parallel Distributed Weka Framework for Big Data Mining Using Spark». In: *2015 IEEE International Congress on Big Data*. New York, NY, USA, June 2015 (cit. on p. 3).

[12] *Singularity*. URL: https://singularity.lbl.gov (cit. on p. 4).

[13] A. Hondroudakis, R. Procter. *The Design of a Tool for Parallel Program Performance Analysis and Tuning*. Edinburgh Parallel Computing Centre. The University of Edinburgh, 1998 (cit. on p. 4).

[14] *TOP500*. URL: https://www.top500.org (cit. on p. 5).

[15] *MareNostrum*. URL: https://www.bsc.es/marenostrum/marenostrum (cit. on p. 5).

[16] *Paraver: a flexible performance analysis tool*. URL: https://tools.bsc.es/paraver (cit. on p. 6).

[17] *Paraver. Tool structure: extracting information from records*. URL: https://tools.bsc.es/paraver/tool_structure (cit. on p. 6).

[18] *Paraver. Trace Generation*. URL: https://tools.bsc.es/paraver/trace_generation (cit. on p. 7).

[19] *Paraver Tracefile Description*. URL: https://tools.bsc.es/doc/1370.pdf (cit. on pp. 7, 39, 42).

[20] *Extrae*. URL: https://tools.bsc.es/extrae (cit. on p. 7).

[21] *PATH and CLASSPATH*. The Java Tutorials, Oracle. URL: https://docs.oracle.com/javase/tutorial/essential/environment/paths.html (cit. on p. 13).

[22] U. Joshi. *How Java thread maps to OS thread?* URL: https://medium.com/@unmeshvjoshi/how-java-thread-maps-to-os-thread-e280a9fb2e06 (cit. on pp. 14, 34).

[23] *Java Native Interface Overview*. The Java Native Interface Programmer's Guide and Specification. URL: https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html#java-native-interface-overview (cit. on p. 17).

[24] *Java Virtual Machine Tool Interface (JVM TI)*. Oracle Docs. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/ (cit. on p. 19).

[25] *Java Virtual Machine Tool Interface (JVM TI) Reference Guide*. Oracle Docs. URL: https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html (cit. on pp. 25, 26, 30, 32).

[26] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. «Aspect-Oriented Programming». In: Palo Alto, CA, USA, 1997 (cit. on p. 28).

[27] *The Anatomy of an Aspect. The AspectJ Language*. Eclipse Foundation. URL: https://www.eclipse.org/aspectj/doc/released/progguide/language-anatomy.html#pointcuts (cit. on p. 28).

[28] *pthread_key_create(3) - Linux man page*. URL: https://linux.die.net/man/3/pthread_key_create (cit. on p. 34).

[29] *GDB: The GNU Project Debugger*. URL: https://www.gnu.org/software/gdb/ (cit. on p. 46).

[30] *Getting started with AspectJ*. URL: https://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html (cit. on p. 50).