Chapter 2

Extrae for JAVA: State of the Art

If it is true that Extrae is not suitable to analyze Java applications, this does not mean that no features were implemented at all. Some basic instrumentation was implemented years ago, but due to its scarce use, it ended to be unmaintained.

This chapter will go through the state of the art of Extrae's instrumentation for Java. It will do that by looking at one at the Extrae's Java example, provided in the Extrae package.

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 the following Code 2.1[1]:

Code 2.1: PiExample.java

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

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

```java
    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 ();
  }
}
```

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

Code 2.2: PiThreaded.java

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

  public PiThreaded (long n, int nthreads)
  {
    m_n = n;
    m_h = 1.0 / (double) n;

    m_threads = new Vector<PiThread>(nthreads);
    for (long i = 0; i < nthreads; i++)
    {
      m_threads.addElement (
        new PiThread (m_h,
          (n/nthreads)*i,
          (n/nthreads)*(i+1)-1)
      );
    }
  }

  public void calculate()
  {
    /* Let the threads run */
    for (int i = 0; i < m_threads.size(); i++)
      (m_threads.get(i)).start();

    /* Wait for their work */
    for (int i = 0; i < m_threads.size(); i++)
    {
      try { (m_threads.get(i)).join(); }
      catch (InterruptedException ignore) { }
    }
  }

  public double result()
  {
    double res = 0.0;
    for (int i = 0; i < m_threads.size(); i++)
    {
      /* reduce the value to result */
      res += (m_threads.get(i)).result();
    }

    return res;
  }
}
```

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

[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 [17]. 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.

in Figure 2.1. As it can be seen, there are not much states shown, but there are several threads detected and traced.

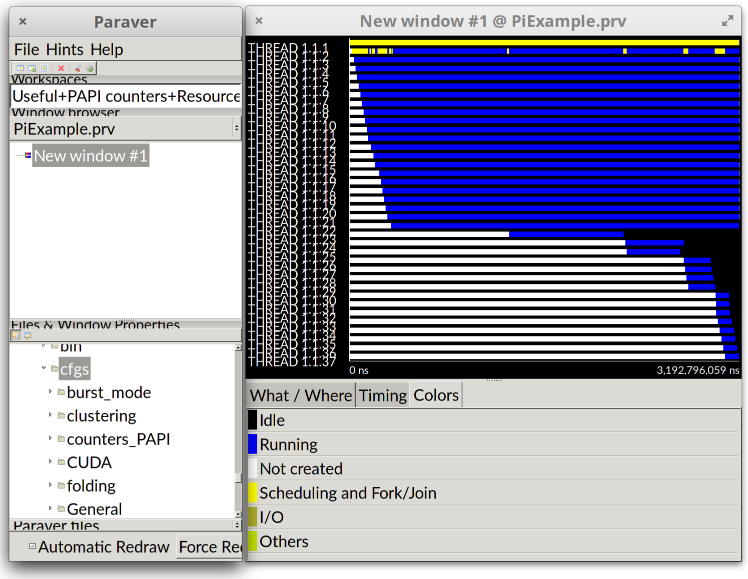

Figure 2.1: PiExample resulting traces

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. The complete `extraej` code is in Appendix B.

Code 2.3: Extract of extraej showing the launching command

```
    LD_PRELOAD=${preload} \
    CLASSPATH=${cp} \
      ${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) [18]. 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. Apparently, out of the pthread creation and termination, no other pthread calls are normally employed in the JVM. That can be easily seen by the previous example, where the synchronization operations present in the Java implementation are not traced.

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: extrae.xml

```xml
<pthread enabled="yes">
  <locks enabled="no" />
  <counters enabled="yes" />
</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 2.2.

The first thread looks exactly the same, while in the second one there are some

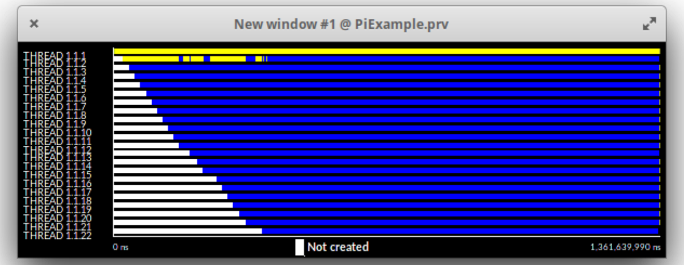

Figure 2.2: Trace of serial PiExample execution

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[3] that can be made is that such threads are inherently launched by the JVM for each execution (Figure 2.3).

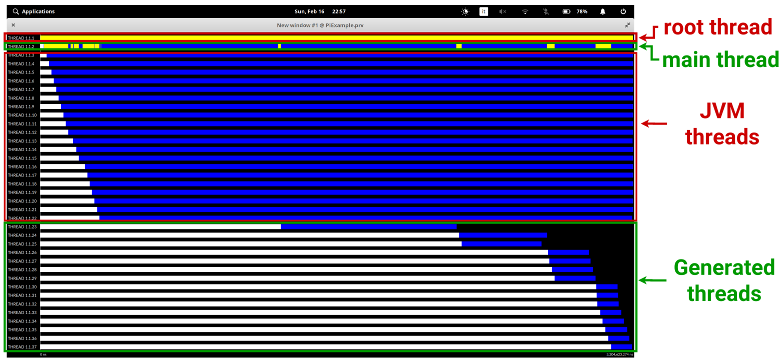

Figure 2.3: PiExample trace explained

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

[3]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[4]. 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 work in this thesis is focused on the mechanisms described above, and to the instrumentation of a framework. These JNI bindings will play a useful role in the later stages of the work.

2.6 Experimental features

Besides the features presented in this chapter, Extrae's reference presented a couple described as "experimental".

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 [20].

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. The problem in this solution was about identification, because the JVM does not

[4]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. [19].

provide the JVM TI with a unique ID for the threads, which is instead required by Extrae 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[5] 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 2.4).

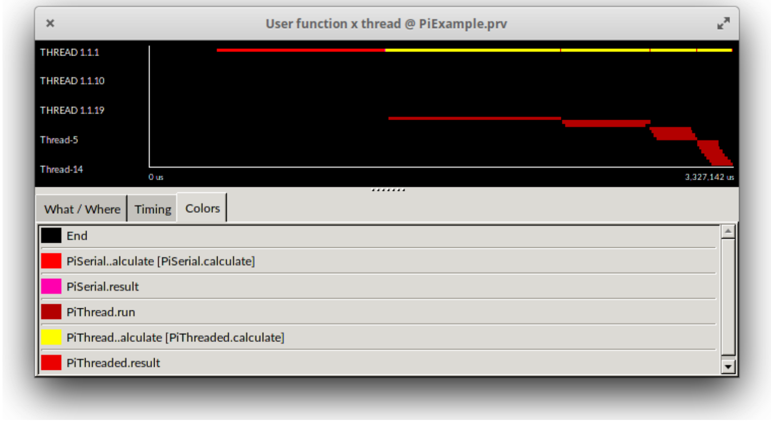

Figure 2.4: Trace showing the user functions for the PiExample program

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

2.7 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.

The second one is the presence of some bugs on the already present Java instrumentation, that make Extrae not working properly with Java programs. 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.