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. [21]

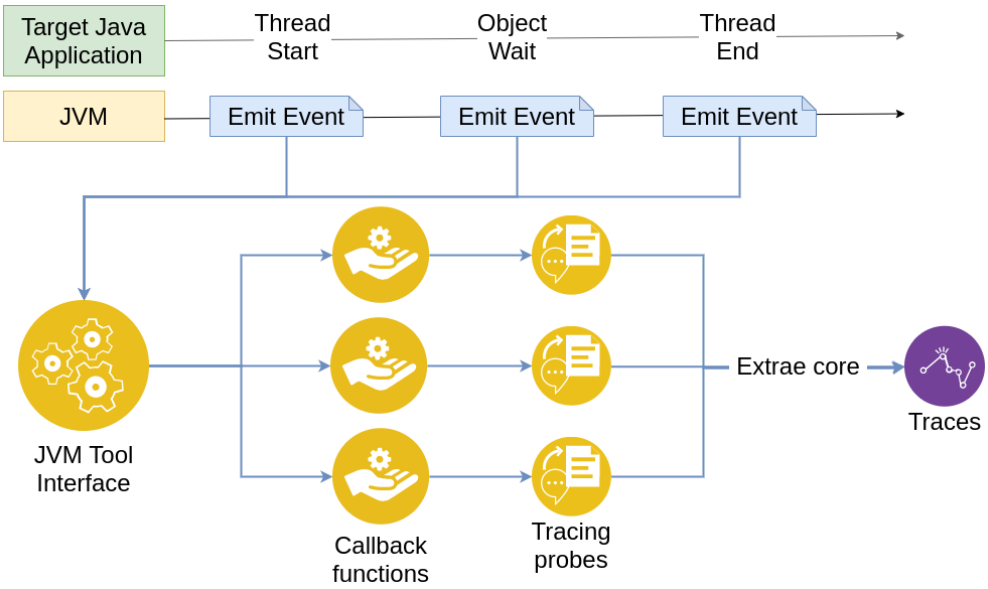

Figure 3.1: Visual explanation of event-driven approach

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 [21], 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.

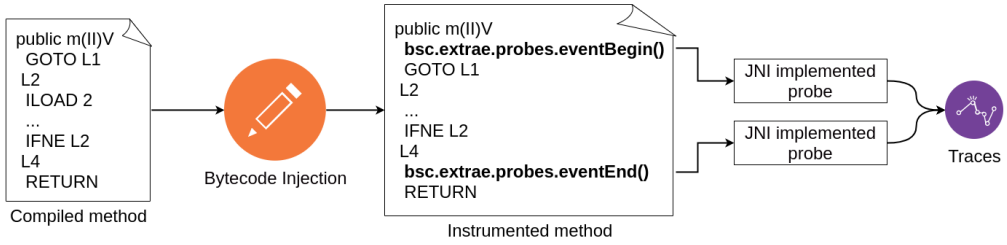

Figure 3.2: Visual explanation of the bytecode instrumentation approach

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 JVM TI, which provides a way to change the name of natively implemented methods, in order to define a wrapper function (similar to what the Linker Preload method does).

[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).

For the instrumentation in Java language, this problem cannot be solved by using the Java instrument API[4]. Instead, this, and 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* [22], 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 [23]. 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

[4]The reason can be found in how the bytecode injection is designed. Indeed, it consists in actually inserting instructions in specific points of the bytecodes, almost like inserting instructions to specific lines of codes. For this reason, it is impossible to inject bytecode in a native method, which is compiled into a binary executable and not into bytecodes.

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 must be instrumented using the JVM TI. Java instrumentation needs JNI methods 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, but also for some native methods instrumentation. For methods tracing instead, AspectJ was preferred to bytecode instrumentation because of its flexibility and ease of use (and maintainability)[5].

[5]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.