A Tracing Technique using Dynamic Bytecode Instrumentation of Java Applications and Libraries at Basic Block Level

Pierre Caserta, Olivier Zendra

A Tracing Technique using Dynamic Bytecode Instrumentation of Java Applications and Libraries at Basic Block Level

Pierre Caserta
Nancy-Université INPL / LORIA
pierre.caserta@loria.fr

Olivier Zendra
INRIA Nancy - Grand Est / LORIA
olivier.zendra@inria.fr

ABSTRACT

Implementing a profiler to trace a program execution is non-trivial. One way to do this on running Java programs is through bytecode instrumentation. Nowadays, tools exist that ease the instrumentation process itself, but as far as we know, none offers an entirely dynamic implementation technique which is able to include the instrumentation of Java Runtime Environement (JRE) classes. In this paper we present the main principles of our technique, which performs such online bytecode instrumentation of both application and JRE classes, at basic block level.

Keywords

Dynamic, Bytecode, Instrumentation, Java, Tracing, Program Analysis, JRE

1. INTRODUCTION

Dynamic analysis provides information on a particular runtime scenario of the software, bringing significant information on program behavior, helping analyzing and understanding it. Among other things, it makes it possible to focus on modules which play a predominant role during the runtime of a program. The main feature of dynamic analysis is that, unlike static analysis which computes all the scenarii, it can capture precise information about the actual runtime of a program. On the other hand, static analysis gives insight about the source code itself and program structure and helps manage its quality. Note that static analysis is more common in the literature than dynamic analysis and many tools are available to extract information from the software source code.

With the emergence of new paradigms such as the Aspect-oriented programming (AOP), rapid prototyping of profilers, tracer, debuggers, and reverse engineering tools can be done more easily [10]. Indeed, dynamically adding new functionalities on running programs, by modifying classes at runtime using bytecode instrumentation, is one convenient way to add behaviors to programs.

We developed the tracing technique we present in this paper by relying on this AOP paradigm to instrument Java programs at strategic join points and extract information from their runtime. The tracing is performed at a fine-grained level, since it records the sequence of basic blocks (control flow) traversed at runtime [1]. In addition, our technique traces both intra and inter-procedural execution flow [7], and includes tracing of JRE classes. Indeed, although many Java tracing techniques focus exclusively on tracing application classes regardless of the JRE, we are convinced that it is necessary in order to fully understand the behavior of a program to be aware of what really happens even when code from the JRE is called, which happens very frequently.

We have aimed at providing this *fine-grained dynamic tracing* of Java software with reasonable performance penalty. Indeed, a difficulty when instrumenting the execution of programs lies in finding an efficient design for the tracer. A very fine-grained technique implies processing a huge amount of information, all the more when JRE classes are traced in addition to application classes.

In this paper, we briefly present the principle of our tracing technique and its implementation. First section 2 explains the tracing technique, which saves static information about the source code of the loaded classes and dynamically extracts the control flow of the runtime. Then section 3 details some interesting properties of our technique. Section 4 gives performance results about our tracer. Section 5 briefly describes how to analyze the trace and the static information, for example in order to compute dynamic metrics. Finally, section 6 concludes and presents future work.

2. TRACING TECHNIQUE

This section explains our tracing technique. First subsection 2.1 reviews our overall technique to trace Java programs. Then subsection 2.2 details the tracer itself. Subsection 2.3 finally discusses the instrumented code.

2.1 Overall view

Figure 1 shows the global functioning of our tracing technique. When a class is loaded by the `ClassLoader`, the Java agent service allows intercepting and instrumenting the bytecode of classes on-the-fly with our `Intrumentor`. The agent is executed in the same JVM (through the `-javaagent` command line option), loaded by the same system class

ICOOOLPS'11, July 26, 2011, Lancaster, UK.

loader, and governed by the same security policy and context as the program. This way all the classes of the JRE and the application can be instrumented. Moreover, classes that are dynamically created by the program are intercepted and instrumented as well. Although other techniques exists, it seems to us that adding only the javaagent option is a convenient way to trace Java programs.

A static analysis of the bytecode is performed during the instrumentation process, extracting static information about basic blocks, methods and classes. In the meantime, instrumentation bytecode is injected into the actual bytecode of the class and adds a new (tracing) behavior without altering the functional semantics of the original bytecode. During the execution phase, the instrumented bytecode provides insight about the control flow to our `Tracer`.

At the end of the runtime (right of Figure 1), all the static information and the dynamic execution trace at basic block level are available in trace files, ready to be processed by any appropriate analyzer. The static information is used to explain all the events that occur during execution. Since metrics are not collected at runtime but computed afterwards, in the second step, the instrumentation tracing slowdown of the program is further limited. Moreover, once the trace is acquired, several kinds of analyses can be performed on the trace such as identifying heavily executed code sequences [6, 2] or calculating worst-case estimates for functions of interest, like in Rapita [3].

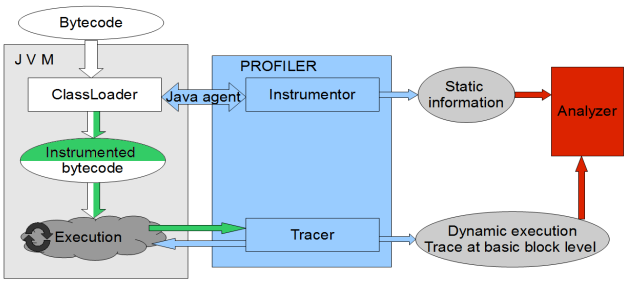

Figure 1: Overall tracing technique

2.2 The tracer

At the core of our technique is a `Tracer` singleton, which can be called by any class of the program. The `Tracer` is the entity that performs all the operations to save the fine-grained runtime trace. In this technique all complex operations of our profiler are performed inside the `Tracer`, which helps us retain control and keep the behavior of the profiled program safe.

Avoiding pollution from the profiler is a crucial task because the profiler code itself relies on JRE classes, which are themselves instrumented. Indeed, JRE classes can be used by both the program and the tracer, so the instrumented classes are a priori shared. With our technique, a simple system of boolean flags that switches tracing on or off can be used avoid having the trace of the tracer in the trace of the program. The principle is trivial but care must be taken because Java programs can be multi-threaded so there is actually a per-thread flag.

2.3 The instrumented code

We wanted the instrumented bytecode to respect the following criteria:

- be as simple and as human-readable as possible
- have the same instrumentation code injected everywhere
- avoid structural changes (no modification of methods signatures, no addition of new methods)

The principle of our technique is to retrieve a complete sequence of events occurring at runtime. When classes are instrumented (at loading), a unique *event number* is associated to each join point (when a method starts, ends, or when a basic block begins). Then, throughout the execution, every time an event occurs the instrumented code informs the `Tracer` using the `notifyEvent` static method and passes an *event number* as argument . Those *event number* are used to identify which part of source code is executed at runtime. The *event number* encodes the information of what happens. Other arguments like the number of the current thread where the event takes place, and the reference on the current instance are also passed to give the possibility of perform reflexion operations on the instance.

So the Java code corresponding to our instrumented bytecode is:

```
Tracer.eventNotifier(eventNumber,
          (int)Thread.currentThread().getId(), this);
```

3. TRACING PROPERTIES

We wanted our tracing technique to have the following properties:

- be easy and ready to use with little configuration (see section 3.1).
- avoid altering the behavior of any program (see section 3.2).
- provide a fine-grained level trace, with with precise control flow (see section 3.3).

3.1 Ease of use

Binder and al. ([4]), in their tool called JP, propose a solution that consists in statically instrumenting core classes of the JRE, whereas all other classes are instrumented at runtime. Their instrumentation adds methods to classes and arguments to methods [10], so the core JRE classes had to be treated as a particular case, with static instrumentation in the `.class` file. Note that our approach is different from other existing techniques which usually add new wrapper methods and add new extra arguments to methods.

On the contrary, since it is completely dynamic, our instrumentation allows transforming only classes which are actually used during the runtime of a specific scenario. Thus there is no need to prepare the classes in advance by altering the original `.class` bytecode files on disk. We believe this is very convenient, because it provides a "zero-config, turn-key and non-invasive" technique.

Some more recent work ([8]) does not perform structural change on classes but still needs to statically modify class `Thread`. Instead of adding new parameters to methods, a new field is added to the `Thread` class and is used as a reference to save the calling context tree. Nonetheless, with

this method, all the JRE classes still have to be statically instrumented, in order to trace native calls. Section 5 explains how we manage to detect native call without using wrappers.

3.2 Same program behavior

The tracer should not alter the behavior of any program and it must only trace the behavior of the program being analyzed, excluding the behavior of the tracer itself. With our technique the tracing is actually performed in the `Tracer` class, so that the instrumented code does not alter the program functional behavior. However, like any Java instrumentation technique which injects bytecodes into classes, it impacts timing and thread scheduling, because the bytecodes coding the instrumentation take time to be executed.

3.3 Fine-grained trace

Our technique performs a fine-grained tracing, at basic block level, which means that it tracks the execution even within methods. This feature provides a call-site-aware tracing [8] which allows the tracing of polymorphism in running programs. In fact, our technique provides enough information to re-simulate precisely the entire runtime flow of the profiled program and thus perform precise dynamic analyses.

4. PERFORMANCE IMPACT

Our technique has been implemented in a tool called VITRAIL JBInsTrace[1]. In this section we show the performance result of our tool on the following benchmarkq: ArgoUML, JEdit, Columba, Ant and the SPECjvm2008 benchmark suite [9].

Our tool has two ways of writing the trace file on disk:

- When memory is constrained, the trace is incrementally written on the disk, at stop-the-world "garbage collector" times. This allows a larger trace to be built up, without having to retain it all in memory all of the time.
- Without memory constraints, the entire trace can be kept in the memory and then dumped on disk at the very end of the execution.

The second technique (no memory constraints) was used for our experimentations. This way it does not bothers the user because the temporal cost is payed only when the program ends. In our figures, we remove the time to write the trace on disk from the total execution time in order to show the actual performance result of the profiler during the execution the program.

Figure 2 shows our performance results. The slowdown factor is 4.9 for ArgoUML, 2.3 for JEdit, 2.4 for Columba and 8 for Ant compiling itself. For most of the benchmarks of the SPECjvm2008 benchmark suite, we found an average slowing factor of 11. Some benchmarks of the SPECjvm2008 benchmark suite are very demanding for the JVM, which probably explains our average slowing factor; this however requires further investigations. These results nonetheless show that the software remains usable with our tracer switched on, which is further confirmed by usability test performed with actual users.

[1]Available at: http://www.loria.fr/~casertap/jbinstrace.html

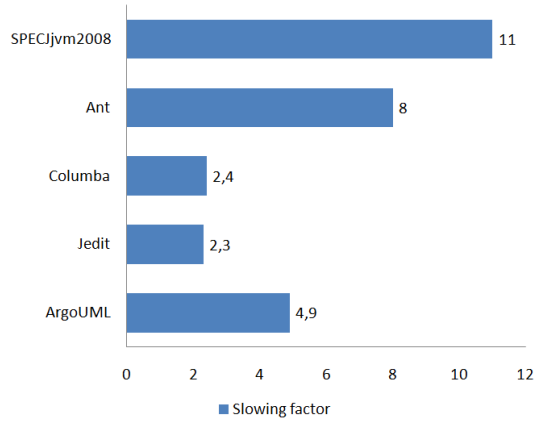

Figure 2: Performance results

We are still working to improve the performance of our tracing technique. However speed is not the main goal: the purpose is to acquire a very detailed trace of the execution, including JRE classes, while keeping the tool easy to use and very flexible regarding the information that can be collected at runtime. This of course implies some penalty in terms of performance.

5. ANALYZING THE TRACE

After being applied to running programs, our tracing technique produces trace files that contain the exact control flow at basic block level, as well as static information about these basic blocks and the global design of the program. The last step is thus to exploit and analyze all this information (see the right part of figure 1).

An analysis technique can be, for each event of the output trace, to follow the dynamic execution and relate each event with the static information saved about each basic block. This exposes which static source code is executed and the dynamic flow of execution of this source code. The whole call stack of the scenario can thus be re-simulated, and very precise dynamic metrics computed. This provides, for example, a mechanism to count the number of executed bytecodes by tracking all the executed basic blocks in the trace, and thanks to the static information sum the size of those basic blocks. This also provides a way to track polymorphism, by comparing the static type of a virtual or interface call site, and the actual type of the receiver or the actual method called in the runtime trace.

The authors of ([5]) have defined several interesting dynamic metrics, which are implemented in our analyzer tool. Dynamic metrics related to e.g program footprint, use of data structures, use of polymorphism, memory accessed, etc. are also easily computable with information provided in our traces.

Some other metrics would nonetheless require additional dynamic logging. For instance, to depict the dynamic type of the calling instance on call sites, Java reflexion must be performed on the calling instance at runtime. We explained in section 2.3 that we pass the reference of the instance by parameter to the `Tracer`. We can thus get the instance dynamic type in the `Tracer`. This shows that our tracing technique can be easily patched with new functionalities without

modifying the whole instrumentation process.

Note that in our technique, it is the callee that informs the trace about calls beginnings and ends, not the caller. However, the tracing of native method calls has to be a bit different, because native methods are not coded in Java, hence cannot be instrumented. Moreover our technique does not add any new method to classes, so using wrapper methods is not possible. As a matter of fact, we use a very simple technique to detect native calls. When runtime is analyzed, the static information about each basic block is followed step by step. When a method call is found in the static information of the basic block, we check whether the corresponding call is present in the actual dynamic trace. If it is not, then we know that this call is a native one.

6. CONCLUSION AND FUTURE WORK

In this paper we presented a tracing technique which is easy to use, can be executed on any Java program, does not alter program behavior nor class files, and provides a fine-grained trace with an acceptable performance penalty, even when including JRE classes.

Our technique makes it possible to trace and analyze what happens at runtime at basic block level. For all we know, no other work has done this fine-grained kind of tracing using only dynamic bytecode instrumentation in Java.

We can trace both program classes and JRE classes because all the dynamically loaded classes are instrumented on-the-fly. This feature gives a very complete trace of the runtime and generates a huge amount of information.

We mainly use our tracer tool to study the dynamic data collected at runtime (gather information and compute metrics) and finally study the habits of object-oriented programmers with a dynamic point of view, so as to better optimize programs (for example: wrt. the use of polymorphism).

One drawback of our technique is that each basic block in instrumented and calls the `Tracer`, which can cause important slowdowns on programs that perform many jumps. Finding a solution to minimize this slowdown is clearly future work to be tackled.

7. REFERENCES

[1] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[2] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[3] A. Betts, N. Merriam, and G. Bernat. Hybrid measurement-based wcet analysis at the source level using object-level traces. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 54–63. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[4] W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144. ACM, 2007.

[5] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, 2003.

[6] J. Larus. Whole program paths. In *ACM SIGPLAN Notices*, volume 34, page 269. ACM, 1999.

[7] D. Melski and T. Reps. Interprocedural path profiling. In *Compiler Construction*, pages 1–99. Springer, 2004.

[8] A. Sarimbekov, P. Moret, W. Binder, A. Sewe, and M. Mezini. Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2011.

[9] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. *Computer Performance Evaluation and Benchmarking*, pages 17–35, 2009.

[10] A. Villazon, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-Oriented Programming. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 63–74, 2009.