Building Java Program Analysis Tools using Javana

Dries Buytaert Jonas Maebe Lieven Eeckhout Koen De Bosschere
ELIS, Ghent University, Belgium
{dbuytaer,jmaebe,leeckhou,kdb}@elis.UGent.be

Abstract

Javana is a tool for creating customized Java program analysis tools. It comes with an easy-to-use instrumentation framework that enables programmers to develop profiling tools that crosscut the Java application, the Java Virtual Machine (JVM) and the native execution layers. The goal of this poster is to demonstrate the power of Javana, using object lifetime computation as an example.

Object lifetime has proven to be useful for analyzing and optimizing the behavior of Java applications. Computing object lifetime is conceptually simple, however, in practice it is often challenging. The JVM needs to be adjusted in numerous ways in order to track all possible accesses to all objects, including accesses that occur through the Java Native Interface (JNI), the standard class libraries, and the JVM implementation itself. Capturing all object accesses through manual instrumentation requires an in-depth understanding of the JVM and its libraries. We show that using Javana is both easier and more accurate than manual instrumentation.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Tracing; D.3.4 [*Processors*]: Run-time Environments

General Terms Experimentation, Measurement, Performance

Keywords Customized Program Analysis Tool, Java, Aspect-Oriented Instrumentation

1. Javana

The popularity of high-level language virtualization software has grown significantly over the recent years with programming environments such as Java and .NET. Because of the tight entanglement between the application and the virtualization software, it becomes difficult to understand the behavior of such applications.

The Javana instrumentation framework provides the end user with both high-level and low-level information [1]. The high-level information relates to the Java application and the VM, such as thread IDs, method IDs, source code line numbers, object IDs, object types, *etc*. The low-level information consists of instruction pointers and memory addresses. Running the Java application of interest within the Javana system along with user-specified instrumentation routines then collects the desired profiles of the Java application.

The Javana system consists of a VM along with a dynamic binary instrumentation tool that runs underneath the VM. The virtual machine communicates with the dynamic binary instrumentation tool through an *event handling* mechanism. The virtual machine informs the instrumentation layer about a number of events, for example when an object is created, moved or collected, or when a method is compiled, re-compiled, *etc*. Using this information, the dynamic binary instrumentation tool builds a *vertical map* that links instruction pointer and memory addresses to high-level language concepts such as objects, methods, etc.

The dynamic binary instrumentation tool that is part of Javana captures all natively executed machine instructions; this includes instructions executed by the application code, instructions that are part of the JVM code, and instructions that are part of the native libraries.

The end result is that Javana knows for all native instructions from what method and thread the instruction comes and to what line of source code the instruction corresponds; and for all accessed memory locations, Javana knows what objects are being accessed. This allows for building a wide variety of profiling tools, such as memory address tracing, vertical cache simulation, object lifetime computation, *etc.*, as shown in [1].

Javana is available for download at the following website: `http://www.elis.ugent.be/javana/`.

2. Object lifetime

Knowing the allocation site and knowing where the object was last used can help a programmer to rewrite the code in order to reduce the memory consumption of the application or even improve overall performance [2].

Computing object lifetimes without Javana is fairly complicated. First, the virtual machine needs to be extended in order to store the per-object lifetime information. Second, special care needs to be taken so that the computed lifetimes do not get perturbed by the instrumentation code. Finally, all object references need to be traced. This is far from trivial to implement. For example, referencing the object's header is required for accessing the Type Information Block (TIB) or vertical lookup table (vtable) on a method call, for knowing the object's type, for knowing the array's length, *etc*. Also, accesses to objects in native methods in the virtual machine or Java standard libraries need to be instrumented manually. Implementing all of this in a virtual machine is very time consuming, error-prone and will likely be incomplete.

Measuring the object lifetime within Javana on the other hand is very easy to do and in addition, it is very accurate because it allows for tracking *all* references to a given object. The skeleton of the instrumentation specification is shown in Figure 1.

The figure illustrates that using Javana, building complex program analysis tools only takes a few lines of code. The language for building Java program profiling tools with Javana is inspired by the Aspect-Oriented Programming (AOP) paradigm. Lines 1 to 4 define a per-object data structure to hold each object's creation time and last access time. Line 5 declares the global time. Time is

Copyright is held by the author/owner(s).
OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

```
0:  #pragma requires object_info

1:  typedef {
2:    unsigned long long creation_time;
3:    unsigned long long last_access;
4:  } object_info_t;

5:  static unsigned long long timestamp = 0;

6:  after object:create (location_t const *loc, type_t const *type, void  **userdata) {
7:    object_info_t ** const objectinfo = (object_info_t**)userdata;
8:    (*objectinfo) = diota_malloc(sizeof(object_info_t));
9:    (*objectinfo)->creation_time = timestamp;
10:   (*objectinfo)->last_access = 0;
11: }

12: before object:access (location_t const *loc, type_t const *type, void  **userdata) {
13:   object_info_t ** const objectinfo = (object_info_t**)userdata;
14:   timestamp++;
15:   (*objectinfo)->last_access = timestamp;
16: }

17: before nonobject:access (location_t const *loc, type_t const *type,  void **userdata) {
18:   timestamp++;
19: }
```

Figure 1. Object lifetime computation tool using Javana.

expressed in terms of the number of memory accesses, and is maintained by the functions below. Lines 6 to 11 define a function that will be called after each object is created; it captures each object's creation time. Lines 12 to 16 define a function that will be called just before an object is accessed. The function is responsible for incrementing the global timestamp counter, and updating the object's last access time. If memory not belonging to an object is accessed, the global timestamp counter is increased by invoking the function specified on line 17 to 19. For more information about the Javana language, we refer to [1].

In order to evaluate the accuracy of object lifetime computations without Javana, we have set up the following experiment. We compute the object lifetimes under two scenarios. The first scenario computes the object lifetime when taking into account all memory accesses as done using Javana. The second scenario uses access information captured by Javana to compute the object lifetime while excluding all object accesses from non-Java code; this excludes all object accesses from native functions. This second scenario emulates current practice of building an object lifetime measurement tool within the virtual machine, without Javana. The results for the SPECjvm98 benchmarks[1] and the DaCapo benchmarks[2] (version beta050224) are shown in Figure 2. The graph shows the percentage of objects for which an incorrect lifetime is computed in current practice, *i.e.*, when not including accesses to objects through JNI functions. We observe large error percentages for a couple of benchmarks, namely fop (4%), antlr (6.5%) and ps (19%). As such, we conclude that current practice of computing object lifetime without Javana can yield incorrect results, and this could be misleading when optimizing code based on these measurements.

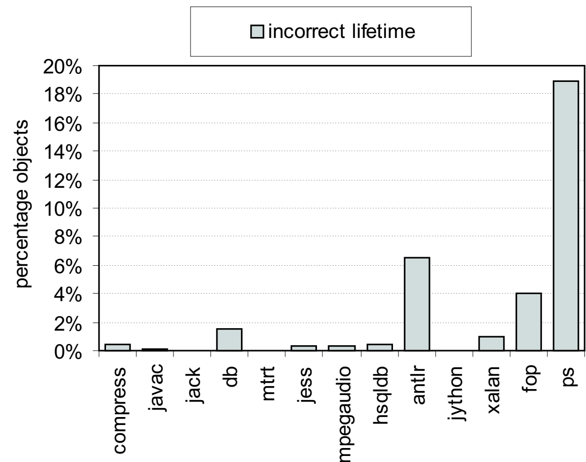

Figure 2. The percentage objects for which a non-Javana instrumentation results in incorrect lifetime computations.

3. Summary

Understanding the behavior of Java applications is non-trivial because of the tight entanglement of the application and the virtual machine. Current practice of building Java program analysis tools involves manually instrumenting the JVM which is both time-consuming and error-prone. In this abstract we demonstrated through the example of object lifetime computation that Javana is an easy-to-use and flexible system for building customized Java program analysis tools.

[1] http://www.spec.org/jvm98/

[2] http://www-ali.cs.umass.edu/DaCapo/gcbm.html

Acknowledgments

Dries Buytaert and Jonas Maebe are supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). This research was also funded by Ghent University.

References

[1] J. Maebe, D. Buytaert, L. Eeckhout, and K. De Bosschere. JAVANA: a system for building customized Java program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, October 2006.

[2] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 104–113, 2001.