JP2 – Collecting Dynamic Bytecode Metrics in JVMs

Aibek Sarimbekov
Walter Binder
University of Lugano, Switzerland
firstname.lastname@usi.ch

Andreas Sewe Mira Mezini
Technische Universität Darmstadt, Germany
lastname@st.informatik.tu-darmstadt.de

Alex Villazón *
Universidad Privada Boliviana (UPB), Cochabamba, Bolivia
avillazon@upb.edu

Abstract

The collection of dynamic metrics is an important part of performance analysis and workload characterization. We demonstrate JP2, a new tool for collecting dynamic bytecode metrics for standard Java Virtual Machines (JVMs). The application of JP2 is a three-step process: First, an online step instruments the application for profiling. Next, the resulting profile is dumped in an appropriate format for later analysis. Finally, the desired metrics are computed in an offline step. JP2's profiles capture both the inter-procedural and the intra-procedural control flow in a callsite-aware calling-context tree, where each node stores, amongst others, the execution count for each basic block of code. JP2 uses portable bytecode instrumentation techniques, is Open Source, and has been tested with several production JVMs.

Categories and Subject Descriptors D.2.8 [*Metrics*]: Performance measures

General Terms Experimentation, Measurement

Keywords Dynamic metrics, Java

1. Introduction

We demonstrate JP2, an open-source tool[1] for portable execution profiling in standard Java Virtual Machines (JVMs). Thanks to plugins that dump the collected profile in an appropriate format, it is easy to set up an evaluation workflow and compute a wide range of dynamic metrics. Recently, JP2 has successfully been used to characterize Scala workloads and to compare their execution behavior with the DaCapo benchmarks [5]; the results of this study are presented in a complementary research paper at OOPSLA 2011 [12].

In terms of implementation, JP2 is a significant improvement over the previous JP profiler [4]. In particular, it improves the completeness and accuracy of profiles in the presence of native methods [11]. Moreover, JP2 adds awareness of both individual callsites [11] and intra-procedural control flow; it thus collects much richer profiles than its predecessor. The collected profiles are complete in the sense that invocations of native methods as well as callbacks from native code into bytecode are all present in the profile. They are also accurate in the sense that for all methods, whether represented by bytecode or by native code, their invocations are counted. Additionally, for methods represented by bytecode, accurate basic-block execution counts are part of the profile.

JP2 arranges the profiles for individual methods in a so-called calling-context tree (CCT) [2], which represents the overall program execution. While rich in information, the resulting profile is therefore conceptually quite simple; the CCT contains only two kinds of objects: nodes representing the calling-contexts and arrays with basic-block execution counts for each context.

2. Step 1: Instrumentation

JP2 applies instrumentation to any method with a bytecode representation, including methods in the standard Java class library. To this end, JP2 uses polymorphic bytecode instrumentation (PBI) [10] to avoid infinite recursions which otherwise may occur when the instrumentation itself invokes methods in the instrumented class library. PBI achieves this using code duplication within method bodies; depending on the control flow, the correct version of the code (either instrumented or original) is executed. A thread-local flag indicates whether execution is at the level of the base program under analysis or at the level of the inserted profiling code. Using PBI requires no structural modifications of classfiles.

For profiling native methods, JP2 relies on native method prefixing, a feature of the JVMTI[2] introduced in Java 6, and statically inserts a bytecode wrapper for each native

* The work presented here was conducted while A. Villazón was visiting University of Lugano, Switzerland.

[1] See `http://jp-profiler.origo.ethz.ch/`.

[2] See `http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/`.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

method [11]. This is the only use of static instrumentation; otherwise all classes get instrumented dynamically, with classes already loaded during JVM bootstrapping being redefined after the bootstrap. If native methods are not of interest, this static instrumentation step becomes optional; all that is required is a Java-6-compatible JVM.

3. Step 2: Profiling and Profile Dumping

Although JP2 collects platform-independent dynamic metrics like method invocations or basic-block execution counts, it can also be used to collect platform-dependent metrics such as elapsed CPU or wallclock time. Momentarily, however, the focus is on dynamic metrics that are independent of both the specific JVM and underlying hardware architecture.

JP2 uses a plugin mechanism for dumping the collected profiles and ships with four different plugins: One pair of plugins uses a text-based output format. Another pair of plugins uses XML-based formats to output either the entire CCT or a dynamic call graph. This flexible plugin mechanism enables the user of JP2 to compute dynamic metrics offline, thus giving an opportunity first to collect the necessary data, then define the metrics of interest, and finally compute them (all from the same profile).

4. Step 3: Metrics Computation

During the live demonstration, JP2 will be applied to several Java and Scala applications, producing detailed profiles. These profiles will be dumped in an XML representation suitable for easy analysis with off-the-shelf tools. We will show several small XQuery scripts [6] that compute various dynamic bytecode metrics of interest, such as method and basic block hotness, callsite polymorphism, and others. The tool demonstration will present all necessary steps for setting up, running, and customizing JP2. Hence, the participants will be able to immediately start using JP2 for their own workload characterization tasks.

5. Related Tools

Dufour et al. [7] presented *J, a tool for collecting dynamic metrics, which relies on the obsolete JVMPI profiling interface.[3] It introduces high overhead and is thus not applicable to large-scale, real-world workloads. Other related tools use sampling techniques that compromise completeness and accuracy of the collected profiles [3, 13]. Finally, many profilers rely on modifications of a particular JVM, such as the Jikes RVM [1] or the Sable VM [8], and are therefore available only for a limited set of environments, such as the profiling framework for multicores devised by Ha et al. [9]. In contrast, JP2 is a tool for collecting complete and accurate dynamic metrics in any standard JVM; moreover, its use incurs only an acceptable overhead [11].

[3] See `http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/`.

References

[1] B. Alpern, D. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, and J. J. Barton. Implementing Jalapeño in Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1997.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2001.

[4] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, 2006.

[6] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, editors. *XQuery 1.0: An XML Query Language*. World Wide Web Consortium, 2nd edition, 2010.

[7] B. Dufour, L. Hendren, and C. Verbrugge. *J: a tool for dynamic analysis of Java programs. In *Companion of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[8] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *In Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2000.

[9] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

[10] P. Moret, W. Binder, and E. Tanter. Polymorphic bytecode instrumentation. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, 2011.

[11] 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 Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2011.

[12] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Dacapo con Scala: Design and analysis of a Scala benchmark suite for the Java virtual machine. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

[13] J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, 2000.