TECHNISCHE
UNIVERSITÄT
DARMSTADT

ULB

Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine

Sarimbekov, Aibek; Moret, Philippe; Binder, Walter et al.

(2011)

DOI (TUprints): https://doi.org/10.25534/tuprints-00014564

Lizenz:

CC-BY-NC-ND 3.0 International - Creative Commons, Namensnennung, nicht kommerziell, keine Bearbeitung

Publikationstyp: Artikel

Fachbereich: 20 Fachbereich Informatik

LOEWE

Quelle des Originals: https://tuprints.ulb.tu-darmstadt.de/14564

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

ELSEVIER

Electronic Notes in Theoretical Computer Science 279 (1) (2011) 61–74

www.elsevier.com/locate/entcs

Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine

Aibek Sarimbekov Philippe Moret Walter Binder[1]

Faculty of Informatics
University of Lugano
Lugano, Switzerland

Andreas Sewe Mira Mezini[2]

Software Technology Group
Technische Universität Darmstadt
Darmstadt, Germany

Abstract

Calling context profiling collects statistics separately for each calling context. Complete calling context profiles that faithfully represent overall program execution are important for a sound analysis of program behavior, which in turn is important for program understanding, reverse engineering, and workload characterization. Many existing calling context profilers for Java rely on sampling or on incomplete instrumentation techniques, yielding incomplete profiles; others rely on Java Virtual Machine (JVM) modifications or work only with one specific JVM, thus compromising portability. In this paper we present a new calling context profiler for Java that reconciles completeness of the collected profiles and full compatibility with any standard JVM. In order to reduce measurement perturbation, our profiler collects platform-independent dynamic metrics, such as the number of method invocations and the number of executed bytecodes. In contrast to prevailing calling context profilers, our tool is able to distinguish between multiple call sites in a method and supports selective profiling of (the dynamic extent of) certain methods. We have evaluate the overhead introduced by our profiler with standard Java and Scala benchmarks on a range of different JVMs.

Keywords: Calling Context Profiling, JP2, Bytecode Instrumentation, Dynamic Metrics

1 Introduction

Calling context profiling is a common profiling technique that helps analyse the dynamic inter-procedural control flow of applications. It is particularly important for understanding and optimizing object-oriented software, where polymorphism

[1] Email: `firstname.lastname@usi.ch`

[2] Email: `lastname@st.informatik.tu-darmstadt.de`

1571-0661 © 2011 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2011.11.006

and dynamic binding hinder static analyses. Calling context profiling hereby collects statistics separately for each calling context, such as the number of method invocations or the CPU time spent in a calling context.

Both the dynamic call graph (DCG) and the Calling Context Tree (CCT) are well-known data structures often used for performance characterization and optimization [1]. The nodes in the respective data structures are associated with profiling information. Such a profile can include a wide range of dynamic metrics, e.g., execution times or cache misses. Platform-independent dynamic metrics such as the number of method invocations or executed bytecodes are of particular interest in the area of performance characterization. These metrics are reproducible,[3] accurate, portable, and comparable [12,4].

Unlike a context-insensitive DCG, a CCT in principle is capable of capturing the *complete* context of a call. Still, CCTs generated by state-of-the-art profilers [17] are missing one key bit of information present in the well-known labelled variant of DCGs: information about the individual site at which a call is made. In other words, while keeping track of numerous methods in entire call chains, many calling context profilers are unable to distinguish between multiple call sites within a single method.

In this paper, we introduce JP2, a call-site-aware profiler for both platform-independent and complete calling context profiling. The profiler is based on portable bytecode instrumentation techniques for generating its profiling data structures at runtime. Besides two counters for the number of method executions and number of executed bytecodes, each calling context tracks the current bytecode position; this enables JP to distinguish between the call sites within a single method.

While several of these features were already present in our earlier JP tool [6], this paper makes several unique contributions:

- A detailed description of JP2, the first call-site aware profiler to capture complete CCTs.
- A description of how JP2 can temporary disable profiling for the current thread without breaking the CCT's structure, hence collecting only appropriate profiles.
- A rigorous evaluation of JP2's performance on 3 virtual machines and 22 Java and Scala benchmarks [7,21].

This paper is structured as follows: Section 2 gives background information on platform-independent dynamic metrics and CCTs. Section 3 describes the tool's design. Section 4 details our performance evaluation on a range of benchmarks and virtual machines. Section 5 discusses related work, before Section 6 concludes.

2 Background

In the following we give a brief overview of both platform-independent dynamic metrics and the Calling Context Tree data structure.

[3] For deterministic programs with deterministic thread scheduling.

2.1 Platform-independent Dynamic Metrics

Most state-of-the-art profilers rely on dynamic metrics that are highly platform-dependent. In particular, elapsed CPU or wallclock time are metrics commonly used by profilers. However, these metrics have several drawbacks: For the same program and input, the time measured can differ significantly depending on the hardware, operating system, and virtual machine implementation. Moreover, measuring execution time accurately may require platform-specific features (such as special operating system functions), which limits the portability of the profilers. In addition, it is usually impossible to faithfully reproduce measurement results.

For these reasons, we follow a different approach that uses only *platform-independent* dynamic metrics [12,4], namely the number of method invocations and the number of executed bytecodes. The benefits of using such metrics are fourfold:

(i) Measurements are *accurate*; profiling itself does not affect the generated profile and will not cause measurement perturbations.

(ii) Profilers can be implemented in a *portable* way; one can execute them on different hardware, operating systems, and virtual machines.

(iii) The profiles made in different environments are *comparable*, as they rely on the same set of platform-independent metrics.

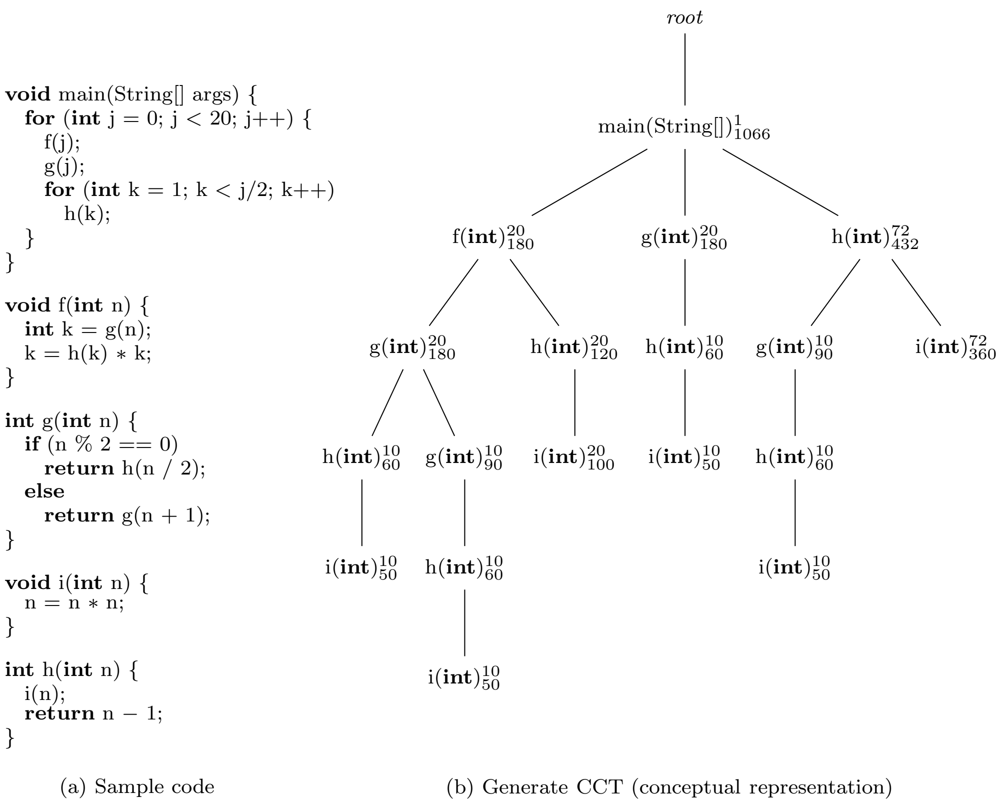
(iv) For deterministic programs, measurements are *reproducible*.

Although information on the number of method invocations is a common metric supported by many available profiling tools, some profilers do not differentiate between different calling contexts or keep calling contexts only up to a pre-defined depth. In contrast, our approach is able to associate both the number of method invocations and the number of executed bytecodes with calling contexts of arbitrary depth.

2.2 The Calling Context Tree (CCT)

The Calling Context Tree (CCT) [1] is a common data structure to represent calling context profiles at runtime [1,2,25,22,26,9]. Each node in a CCT corresponds to a calling context and keeps the dynamic metrics measured for that particular calling context; it also refers to the method in which the metrics were collected. The parent of a CCT node represents the caller's context, while the children nodes correspond to the callee methods. If the same method is invoked in distinct calling contexts, the different invocations are thus represented by distinct nodes in the CCT. In contrast, if the same method is invoked multiple times in the same calling context *and* from the same call site, the dynamic metrics collected during the executions of that method are kept in the same CCT node. The CCT thus makes it possible to distinguish dynamic metrics by their calling context. This level of detail is useful in many areas of software engineering such as profiling [1], debugging [3], testing [20], and reverse engineering [15].

It should be noted that the data structure itself does not impose any restrictions on the number and kind of dynamic metrics kept in the CCT nodes; in particular,



(a) Sample code (b) Generate CCT (conceptual representation)

Fig. 1. Sample Java code and the CCT generated for one execution of method main(String[]). As dynamic metrics, each CCT node stores the number of method invocations (^{m}) and the number of executed bytecodes ($_{n}$) in the corresponding calling context.

these metrics may be platform-dependent (CPU time, number of cache misses) or platform-independent (number of method invocations [4], number of executed bytecodes, number of object allocations). In the following, we will restrict the discussion to two platform-independent metrics: the number of method invocations and the number of executed bytecodes. Fig. 1 exemplifies such a CCT data structure, which stores both metrics.

CCTs are most useful if they faithfully represent overall program execution. We thus require that a *complete* CCT contains all method invocations made after an initial JVM bootstrapping phase [5], where either the caller or the callee is a Java method, i.e., a method not written in native code. The following method invocations must therefore be present within the CCT:

(i) A Java method invoking another Java method.

(ii) A Java method invoking a **native** method.

(iii) Native code invoking a Java method (e.g., callback from native code into byte-

[4] In this paper we do not distinguish between methods and constructors; 'method' denotes both 'methods' and 'constructors'.

[5] At the latest, this phase ends with the invocation of the program's main(String[]) method.

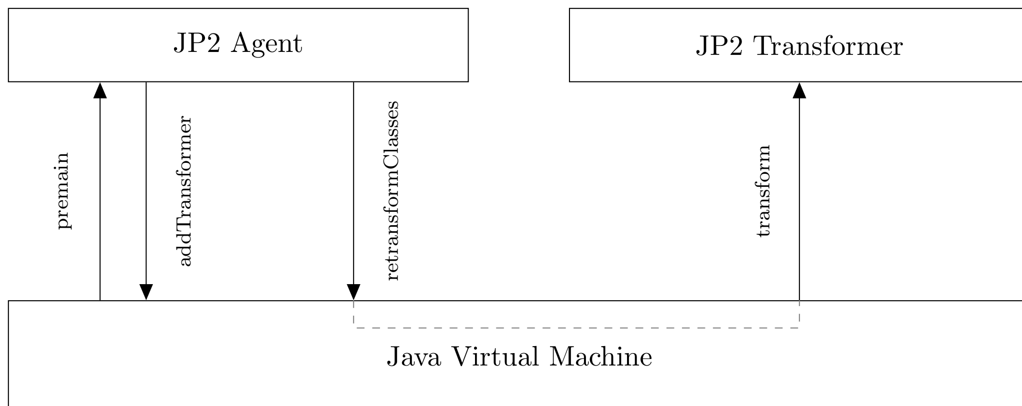


Fig. 2. The architecture of JP2.

code through the Java Native Interface (JNI), class loading, or class initialization)

Regarding method invocation through reflection, the above definition of completeness implies that any method called through reflection (Method.invoke(...), Constructor.newInstance(...)) must be represented in the CCT.

There are several variations of the CCT supported by our tool. For instance, calls to the same method from different call sites in a caller may be represented by the same node or by different nodes in the CCT. Moreover, in the case of recursion, the depth of the CCT may be unbounded, representing each recursive call to a method by a separate CCT node, or alternatively recursive calls might be stored in the same node, limiting the depth of the CCT and introducing back-edges into the CCT [1].

3 Tool Design

In this section we describe the design and the architecture of our tool. First, Section 3.1 discusses both the design and the weaknesses of a previous version of the tool. Next, Section 3.2 presents our new design implemented in the JP2 profiler. Finally, Section 3.3 explains how JP2 deals with native methods.

3.1 Old Design (JP)

Our previous profiler, JP [17,6], is based on the generic bytecode instrumentation framework FERRARI [5], which allows us to statically instrument the Java class library and apply load-time instrumentation to all other classes. JP extends method signatures in order to pass a CCT node reference from the caller to the callee as an argument. Therefore, each method invocation keeps a reference to its corresponding CCT node in a local variable. Furthermore, for each method a static field is added to hold the corresponding method identifier and the class initializer is instrumented to allocate those method identifiers. For compatibility with native code, wrapper methods with unmodified signatures are added to allow native code, which is not

aware of the additional argument, to invoke the instrumented method through the JNI.

This approach introduces compatibility problems with recent JVMs because some methods cannot be overloaded in this fashion. Moreover, the additional stack frames introduced by wrapping methods can break stack introspection. Furthermore, static instrumentation of the Java class library is time consuming and inconvenient to the user. Finally, JP is unable to distinguish between different call sites and one cannot selectively enable or disable metrics collection for a certain calling context, e.g., for a benchmark's harness.

3.2 New Design (JP2)

Fig. 2 depicts the architecture of JP2, our revised design which addresses all of JP's weaknesses mentioned above. JP2 includes two main components with the following responsibilities:

(i) The *JP2 Agent* is a Java programming language agent; it initializes JP2 upon JVM startup.

(ii) The *JP2 Transformer*, which is based on the ASM bytecode engineering libary [6], is responsible for the necessary bytecode instrumentation.

Upon startup, the JVM invokes the premain() method of the JP2 Agent before any application code is executed. The agent registers the transformer using the standard java.lang.instrument API. Through this API, the agent then triggers retransformation of classes that have already been loaded during JVM startup. During retransformation, the JVM calls back the transformer, which instruments the classes. The JP2 Transformer receives transform() requests both for classes to be retransformed, as well as for newly loaded classes (see Fig. 2).

In contrast to JP, JP2 does not need to extend any method signatures in order to pass a CCT node; instead, it uses thread-local variables to store references to them. Moreover, instead of adding static fields storing the method identifiers, JP2 uses string constants which simply reside in the class file's constant pool; thus, there is no need for extending the class initializer anymore. Fig. 3 illustrates the instrumentation the JP2 Transformer applies. Depicted to the left is the method f() before transformation; depicted to the right is the corresponding instrumented version. [7] The transformer inserts invocations to static methods in class JP2Runtime shown in Fig. 4, which are explained below.

setBI(int callerBI), getBI() Store the caller's bytecode position in a thread-local variable, respectively load the stored bytecode position (BI) from the thread-local variable.

setCurrentNode(CCTNode n), getCurrentNode() Store the current thread's current CCT node in a thread-local variable, respectively load the

[6] See http://asm.ow2.org/.

[7] To improve readability, all transformations are shown in Java-based pseudo code, although JP2 works at the JVM bytecode level.

```
BI      void f() {



          while (true) {

            if (i <= 10) {

5:
              h();
7:
              g(i);
              ++i;
            } else {

              return;
            }
          }




        }
```

(a) Before Instrumentation

```
void f() {
  int callerBI = JP2Runtime.getBI();
  CCTNode caller = JP2Runtime.getCurrentNode();
  CCTNode callee = caller.profileCall("f()", callerBI);
  try {
    callee.profileBytecodes(2);
    while (true) {
      callee.profileBytecodes(3);
      if (i <= 10){
        callee.profileBytecodes(5);
        JP2Runtime.setBI(5);
        h();
        JP2Runtime.setBI(7);
        g(i);
        ++i;
      } else {
          callee.profileBytecodes(1);
          return;
        }
    }
  } finally {
    JP2Runtime.setCurrentNode(caller);
    JP2Runtime.setBI(callerBI);
  }
}
```

(b) After Instrumentation

Fig. 3. Example of Java code instrumented by JP2.

```
public class JP2Runtime {
  public static int getBI() {...}
  public static void setBI(int callerBI) {...}
  public static CCTNode getCurrentNode() {...}
  public static void setCurrentNode(CCTNode n) {...}
}

public interface CCTNode {
  CCTNode profileCall(String methodID, int callerBI);
  void profileBytecodes(int i);
}
```

Fig. 4. Runtime classes used by JP2.

current CCT node from a thread-local variable.

Hereby, CCTNode is an interface shown in Fig. 4, whose methods perform the following functions:

profileCall(String methodID, int callerBI) Return the callee of the method in the CCT; if there is no such node, register it.

profileBytecodes(int) Update the counter keeping the number of executed bytecodes.

It is crucial to restore the caller's bytecode position in the **finally** block because class loading and class initialization may be triggered between a call to setBI(**int**) in a caller and the subsequent method call, which may in turn update the thread-local variable.

(a) Before wrapping

```
native boolean foo(int x);
```

(b) After wrapping

```
boolean foo(int x) {
  return wrapped_foo(x);
}

native boolean wrapped_foo(int x);
```

Fig. 5. Example of a native method wrapped by JP2.

JP2 counts bytecodes per basic block using the same algorithm as JP [6]: only bytecodes that may change the control flow non-sequentially (i.e., jumps, branches, return of method or JVM subroutine, exception throwing) end a basic block. This algorithm creates rather large basic blocks, such that the number of updates to the bytecode counter is kept low. This reduces runtime overhead without significantly affecting accuracy [6].

JP2 provides a mechanism to temporarily disable the execution of instrumentation code for each thread. Assume that instrumentation code itself uses methods from the Java class library, which has already been instrumented. This will cause infinite recursions. To sidestep this issue, JP2 uses code duplication within method bodies in order to keep the non-instrumented bytecode version together with the instrumented code, and inserts a conditional upon the method entry in order to select the version to be executed. [16]

JP2 allows to selectively activate and deactivate the collection of dynamic metrics for each thread without breaking the structure of the CCT. In Section 4 we use this feature to collect proper profiles only for the execution of the benchmarks, excluding the execution in the harness and the JVM's startup and shutdown sequences.

3.3 Native Methods

To gather complete profiles, JP2 has to keep track of all native method invocations as well as callbacks from those native methods. Since native methods do not have any bytecode representation, they cannot be instrumented directly. As illustrated by Fig. 5, JP2 thus adds simple wrapper methods with unmodified signatures. Native method prefixing [23], a functionality introduced in Java 6, is used to rename native methods and introduce a bytecode implementation with the name of the original native method. However, certain limitations prevent JP2 from applying the transformation at runtime to classes loaded during JVM bootstrapping. While class redefinition may change method bodies, the constant pool and attributes, it cannot add, remove or rename fields or methods, and change the signatures of methods. Therefore, JP2 is accompanied by a static tool, whose sole purpose is to add those wrappers to the Java class library.

This is the only circumstance under which JP2 has to resort to static instrumentation, which should be done before any dynamic instrumentation. Later, the wrapped Java class library is added at the beginning of the boot class path. Since the JVM needs to invoke native methods upon bootstrapping, JP2 has to make it

aware of the added prefix. Therefore, a JVMTI agent, which only informs the JVM about the prefix, needs to be passed as a command line option to the JVM.

4 Evaluation

In order for a profiling tool to be universally useful, it has to be *stable* and *portable*. Furthermore, it must not impose prohibitive measurement overhead, i.e., slow down the application by orders of magnitude. In our evaluation we show that JP2 has all three properties; when running a diverse selection of benchmarks on a set of production JVMs it imposes acceptable runtime overhead.

To this end, we have evaluated the runtime overhead incurred by JP2 using two different benchmark suites: the DaCapo 9.12-bach benchmark suite [7] and a DaCapo-based benchmark suite consisting of Scala programs, which is under active development by one of the authors [21]. In either case, the measurements exclude the startup and shutdown of both JVM and benchmark harness. JP2 itself has also been configured to collect dynamic metrics only for the benchmark proper, of whose iterations it is notified using the callback mechanisms provided by the benchmark harness (`Callback`).

To both show that JP2 is portable and to assess the effect a given JVM can have on the runtime overhead incurred by JP2, we have performed all measurements using three configurations representative of modern production JVMs: the HotSpot Server VM [8], the HotSpot Client VM, [8] and the JRockit VM [9]. All benchmarks have been run on a 2.33 GHz Core 2 Duo dual core E6550 processor with 2 GB of main memory, 32 KB L1 data and instruction caches, and 4096 KB L2 cache; its entire main memory has been available to the JVM (`-Xmx2G`). During benchmarking, the computer was operating in single-user mode under Ubuntu Linux 9.10 (kernel 2.6.31).

Fig. 6 depicts the overhead incurred by JP2 during the first iteration of the 14 DaCapo 9.12 benchmarks when using the three virtual machines mentioned above. As can be seen, on most virtual machines the overhead is moderate; the slowdown is less than one order of magnitude. The only exception from this is the HotSpot Client VM. Here, JP2 incurs significantly higher overheads, as the VM's client compiler [13] copes less well than the server compiler [19] with the instrumentation inserted by JP2. But as JP2 produces mostly platform-independent profiles, it is often possible to reduce overheads to acceptable levels simply by choosing a different virtual machine that copes better with JP2's instrumentation; the resulting CCTs will differ only within the platform-specific part of the given Java class library, not within the application.

Also, part of the runtime overhead is incurred by JP2 only upon class-loading, i.e., when newly loaded classes are instrumented. Fig. 7 illustrates this fact; the absolute overhead diminishes over the course of several iterations of a benchmark or during long-running applications. The relative overhead, however, increases, as the

[8] JRE build 1.6.0_22-b04, JVM build 17.1-b03
[9] JRE build 1.6.0_20-b02, JVM build R28.0.1-21-133393-1.6.0_20-20100512-2126-linux-ia32

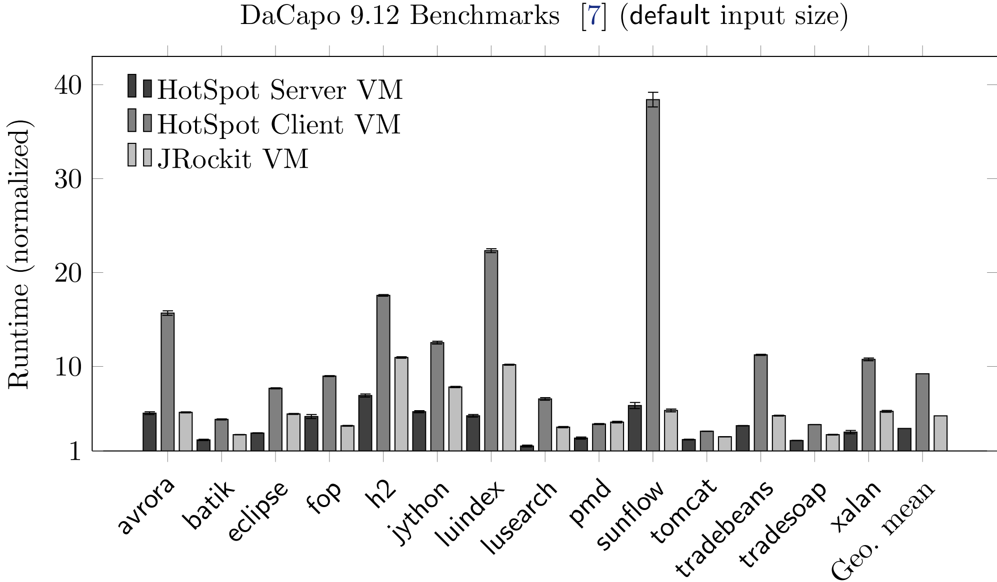


Fig. 6. Runtime overhead (5 invocations, arithmetic mean ± sample standard deviation) incurred by JP2 during the first iteration of 14 Java benchmarks on 3 different JVMs.

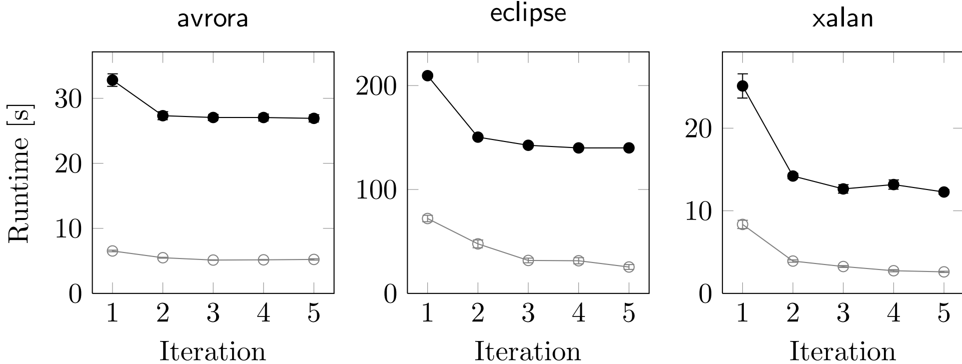


Fig. 7. Runtime (5 invocations, arithmetic mean ± sample standard deviation) with (—●—) and without (—○—) JP2 over serveral iterations of three benchmarks on the HotSpot Server VM.

more advanced optimizations performed by the just-in-time compiler during later iterations are hindered by the instrumentation inserted by JP2.

Fig. 8 depicts the overhead incurred by JP2 on a set of Scala benchmarks. As the CCTs generated for several benchmarks (kiama, scalac, and scaladoc) exceed the heap's capacity of 2 GB, only the small input size has been used for those benchmarks. But when compared to the Java benchmarks of Fig. 6, the overhead incurred by JP2 on the Scala benchmarks is remarkably similar: For only two benchmarks (scalaxb, tmt), the HotSpot Server VM, which performs best with JP2 on the Java benchmarks, experiences more than moderate performance degradation on the Scala benchmarks.

Fig. 9 shows two key properties of the CCTs generated for various benchmark programs: the number of unique methods called and the number of CCT nodes

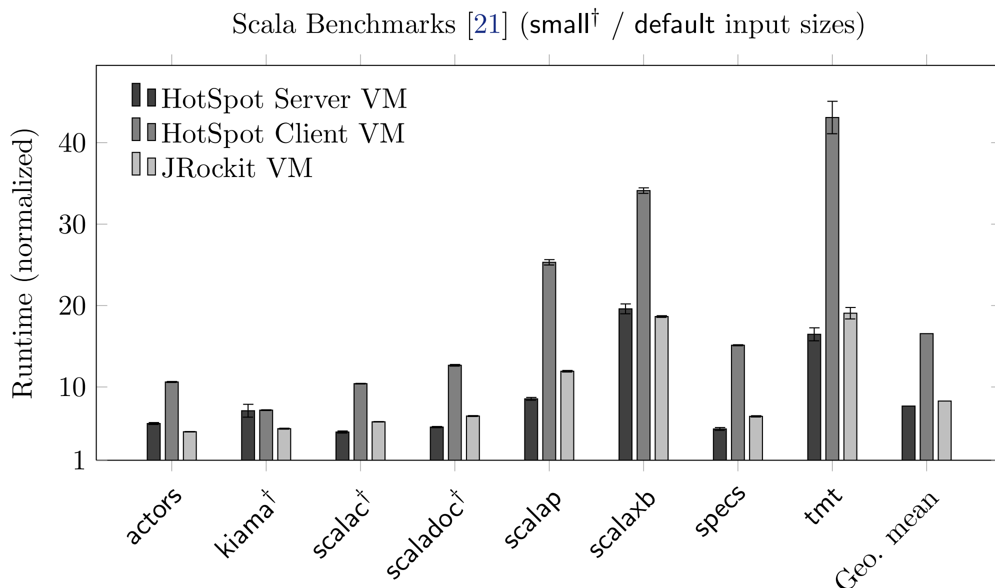


Fig. 8. Runtime overhead (5 invocations, arithmetic mean ± sample standard deviation) incurred by JP2 during the first iteration of 8 Scala benchmarks on 3 different JVMs.

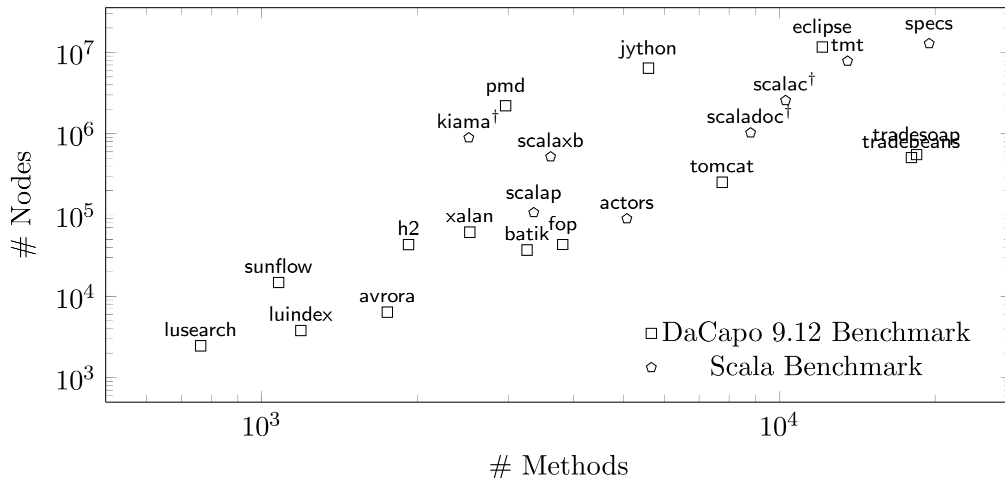


Fig. 9. Number of called methods and number of generated CCT nodes for different benchmarks. (For some Scala benchmarks, only the small† input size was measured.)

that result therefrom. As JP2 has to keep the CCT in memory, benchmarks with millions of CCT nodes (the Java benchmarks eclipse, jython, and pmd; the Scala benchmarks scalac, specs, and tmt) naturally put additional pressure on the JVM's garbage collector, which has to trace a large data structure that never dies till VM shutdown. Nevertheless, as Fig. 9 shows, JP2 is able to deal with large programs consisting of tens of thousands of methods.

5 Related Work

Calling context profiling has been explored by many researchers. Existing approaches that create accurate CCTs [22,1] suffer from considerable overhead. Sampling-based profiles promise a seemingly simple solution to the problem of large profiling overheads. However, as Mytkowicz et al. have recently shown [18], implementing sampling-based profilers correctly such that the resulting profiles are at least "actionable" if not accurate is an intricate problem which many implementations fail to solve. JP2 sidesteps this issue by focussing on machine-independent metrics, which it measures both accurately and with moderate profiling overhead.

Dufour et al. [11] present a variety of dynamic metrics, including bytecode metrics, for selected Java programs, such as the SPEC JVM98 benchmarks [24]. They introduce a tool called *J [12] for the metrics computation. *J relies on the JVMPI [14], a profiling interface for the JVM, whose use is known to cause high overhead when recording, e.g., method entry and exit events like JP2 does [10]. Furthermore, JVMPI is an interface no longer supported as of the Java 6 release (late 2006).

The NetBeans Profiler [11] integrates Sun's JFluid profiling technology [10] into the NetBeans IDE. JFluid exploits dynamic bytecode instrumentation and code hotswapping in order to turn profiling on and off dynamically, for the whole application or just a subset of it. However, this tool needs a customized JVM and is therefore only available for a limited set of environments. In contrast, JP2 works with any standard JVM without customization.

The Probabilistic Calling Context (PCC) approach due to Bond et al. [9] continuously maintains a probabilistically unique value representing the current calling context. As this value can be efficiently computed, the approach causes rather low overhead, if supported by a customized virtual machine. But due to its probabilistic nature PPC does not always produce completely accurate profiles. Recent research, however, has shown that is often possible to reconstruct a significant amount of context offline [8].

6 Conclusion

In this paper we presented JP2, a new tool for complete platform-independent calling context profiling. JP2 relies on bytecode transformation technique in order to create CCTs with platform-independent dynamic metrics, such as the number of method invocations and the number of executed bytecodes. In contrast to prevailing profilers, JP2 is able to distinguish between multiple call sites in a method and supports selective profiling of certain methods. We have evaluated the overhead caused by JP2 with standard Java and Scala benchmarks on a range of different JVMs.

[10] For the fop Java benchmark, e.g., *J increases runtime by a factor of 33.

[11] See http://profiler.netbeans.org/.

Acknowledgement

This work has been supported by the Swiss National Science Foundation and by CASED (http://www.cased.de/www.cased.de).

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[3] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In J. Vitek, editor, *ECOOP '08: Proceedings of the 22th European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 542–565, Paphos, Cyprus, 2008. Springer-Verlag.

[4] W. Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 178–194, Tsukuba, Japan, Nov. 2005. Springer Verlag.

[5] W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM.

[6] 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.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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, 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 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[8] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 13–24, New York, NY, USA, 2010. ACM.

[9] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming, systems and applications*, pages 97–112, New York, NY, USA, 2007. ACM.

[10] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.

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

[12] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.

[13] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot^{TM} client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5:7:1–7:32, May 2008.

[14] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-99)*, pages 229–240, Berkeley, CA, May 3–7 1999. USENIX Association.

[15] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[16] P. Moret, W. Binder, and E. Tanter. Polymorphic bytecode instrumentation. In *International Conference on Aspect-Oriented Software Development '11*, Porto de Galinhas, Pernambuco, Brasil, March 21-25 2011. Publication forthcoming.

[17] P. Moret, W. Binder, and A. Villazón. CCCP: Complete calling context profiling in virtual execution environments. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 151–160, Savannah, GA, USA, 2009. ACM.

[18] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM.

[19] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[20] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *FASE '05: Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science (LNCS)*, pages 282–297, April 2005.

[21] A. Sewe. Scala $\stackrel{?}{\equiv}$ Java mod JVM. In *Proceedings of the Work-in-Progress Session at the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010)*, volume 692 of *CEUR Workshop Proceedings*, 2010.

[22] J. M. Spivey. Fast, accurate call graph profiling. *Software: Practice and Experience*, 34(3):249–264, 2004.

[23] Sun Microsystems, Inc. JVM Tool Interface (JVMTI), Version 1.0. Web pages at http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/, 2004.

[24] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at http://www.spec.org/osg/jvm98/, 1998.

[25] J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, June 2000.

[26] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM.