Portable and Accurate Collection of Calling-Context-Sensitive Bytecode Metrics for the Java Virtual Machine

Aibek Sarimbekov
University of Lugano
aibek.sarimbekov@usi.ch

Andreas Sewe
Technische Universität Darmstadt
sewe@st.informatik.tu-darmstadt.de

Walter Binder Philippe Moret
University of Lugano
{walter.binder, philippe.moret}@usi.ch

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

Mira Mezini
Technische Universität Darmstadt
mezini@st.informatik.tu-darmstadt.de

Abstract

Calling-context profiles and dynamic metrics at the bytecode level are important for profiling, workload characterization, program comprehension, and reverse engineering. Prevailing tools for collecting calling-context profiles or dynamic bytecode metrics often provide only incomplete information or suffer from limited compatibility with standard JVMs. However, completeness and accuracy of the profiles is essential for tasks such as workload characterization, and compatibility with standard JVMs is important to ensure that complex workloads can be executed. In this paper, we present the design and implementation of JP2, a new tool that profiles both the inter- and intra-procedural control flow of workloads on standard JVMs. JP2 produces calling-context profiles preserving callsite information, as well as execution statistics at the level of individual basic blocks of code. JP2 is complemented with scripts that compute various dynamic bytecode metrics from the profiles. As a case-study and tutorial on the use of JP2, we use it for cross-profiling for an embedded Java processor.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement techniques; D.2.8 [*Software Engineering*]: Metrics—Performance measures; B.8.2 [*Performance and Reliability*]: Performance Analysis and Design Aids

General Terms Measurement, Performance

Keywords Calling Context Tree, bytecode instrumentation, Java Virtual Machine, cross-profiling

1. Introduction

The availability of comprehensive, calling-context sensitive dynamic metrics helps improve correctness and speed of software engineering tasks such as program comprehension and reverse engineering [26]. Furthermore, characterization of Java workloads requires exact statistics about the executed bytecodes representing overall program execution. In addition, many code optimization tasks benefit from detailed profiling information.

While there is undoubtedly need for profiles that capture overall program execution on any standard, state-of-the art Java Virtual Machine (JVM) and represent both the inter- and intra-procedural control flow, available profiling tools for the JVM produce only incomplete information for the aforementioned tasks. Furthermore, they either incur prohibitive overhead (both in terms of execution time and space needed for the profiles in memory or on secondary storage) or require a special, modified JVM. For instance, the dynamic metrics collection tool *J [19], which has been used for characterizing the SPEC JVM98 benchmarks,[1] incurs excessive overhead and is hardly applicable to recent, more complex Java workloads such as the DaCapo benchmarks [11]. A large body of related work on profiling therefore focuses on sampling techniques that can be useful for detecting hotspots and performance bottlenecks in programs, but fail to give a complete and accurate view of overall program execution [3, 6, 34, 35]. Profiling frameworks such as the Arnold/Ryder framework for efficient sampling [3] or the concurrent dynamic-analysis framework of Ha et al. [20] rely on modifications of research VMs such as the Jikes RVM [1] and are not available on production JVMs such as Oracle's HotSpot VM. While the calling-context profiler JP presented in prior work [5, 8, 24] is available for production JVMs, it lacks support for intra-procedural profiling and makes use of several fragile instrumentation techniques that fail on some recent production JVMs.

In this paper we present JP2, a new profiler that produces complete and accurate calling-context profiles on production JVMs. In such a calling-context profile individual calls are distinguished by the context they occur in, i.e., by the entire call stack that led to the call. (See Section 2 for an example.) In contrast to the profiles generated by the former profiler JP, the profiles produced by JP2 keep track of different callsites and thus directly reflect the call stack. Furthermore, JP2 provides exact execution statistics for each basic block of code. The instrumentation applied by JP2 to collect these

PPPJ '11, August 24–26, 2011, Kongens Lyngby, Denmark.

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

profiles avoids structural modifications of classfiles as much as possible in order to achieve compatibility with production JVMs.

Completeness of the profile means that after an initial JVM bootstrapping phase (which ends with the invocation of the application's main method), the invocation of every method[2] is profiled. This definition of profile completeness implies that invocations of native methods from bytecode as well as call-backs from native code into bytecode need to be tracked as well. Furthermore, certain bytecodes like instanceof or new may trigger method invocations to load or initialize a class. These implicit invocations also have to be tracked. Accuracy of the profile means that it faithfully represents program execution and implies, for example, that method invocation counters and callsite information are correctly tracked.[3] The intra-procedural control flow is provided for each non-native method; for each basic block, an execution counter is kept. JP2 supports pluggable basic block analysis algorithms [8].

The resulting profile is represented as a single Calling Context Tree (CCT) [2] for all threads executing in the JVM. JP2 uses a thread-safe, non-blocking data structure to store the CCT at runtime. This data structure has been tuned to minimize the number of allocated objects. For each calling-context, only two objects are allocated: one object representing the calling-context as a node in the CCT, and one array keeping track of the execution counts of each basic block in the corresponding method.

JP2 offers several plugins to serialize the CCT upon program termination. The plugin used in this paper stores the CCT in an XML format. Together with the stored classfiles, converted to XML by the ASM bytecode engineering library,[4] various dynamic metrics can be computed using XQuery [12] scripts by cross-referencing between CCT and classfiles.

To show the versatility of our approach and as a tutorial for computing dynamic metrics with the help of JP2 and XQuery, we use JP2 for cross-profiling embedded Java applications [9, 10]. That is, we illustrate XQuery scripts that take the profile produced in any host environment on any standard JVM and estimate the CPU cycles the same program run would consume on an embedded Java processor that serves as cross-profiling target. In this example, we exploit both the intra-procedural and the inter-procedural information conveyed in the profile.

The original, scientific contributions of this paper are four-fold:

1. We discuss the instrumentation performed by JP2.
2. We explain the thread-safe, non-blocking data structure employed by JP2.
3. We illustrate the use of XQuery scripts to compute dynamic metrics from JP2's output. Cross-profiling for an embedded Java processor serves us as a case study.
4. We present a detailed evaluation of profiling overhead with the DaCapo benchmarks, exploring the different sources of overhead.

JP2 is Open-Source, licensed under the GNU General Public License. It requires JDK 1.6 or higher (if native-method prefixing is enabled) and has been tested with different versions of Oracle's JDK 1.6 under Linux, Mac OS X, and Windows. The profiler and associated tools are available for download from the project's website:

`http://jp-profiler.origo.ethz.ch/`

[2] In this paper 'method' stands for 'method or constructor.'

[3] For call-backs from native code into bytecode, callsite information is not available, but must be denoted by a special value (e.g., -1).

[4] See `http://asm.ow2.org/`.

An initial paper on JP2 has been previously presented at the ByteCode'11 workshop [27]. However, our previous work considered neither intra-procedural profiling at the basic block level, nor thread-safety of the CCT data structure, nor the computation of dynamic metrics with XQuery scripts, nor a detailed analysis of the sources of overhead.

This paper is structured as follows: First, Section 2 describes the instrumentation scheme applied by JP2, before Section 3 provides details on the thread-safe, non-blocking data structure used. Next, Section 4 briefly discusses how profiles are serialized. Section 5 presents an extensive case study on using JP2 and XQuery for cross-profiling an embedded Java processor. Section 6 evaluates the performance of JP2 and its various features. Finally, Section 7 discusses related work, before Section 8 concludes.

2. Instrumentation

In this section, we discuss the instrumentation scheme applied by JP2. For illustration, let us consider the example class Demo shown in Figure 1. While simple in nature, the example serves to illustrate several interesting cases, including a polymorphic callsite. In the example, the BP comments specify the bytecode positions[5] of callsites in the compiled bytecode.

A schematic representation of the CCT corresponding to one execution of the main method is shown in Figure 2. For each node in the CCT, we store the method identifier, the bytecode position of the callsite, the number of method invocations, and the basic-block execution counts. As we can see in Figure 2, the root node has the (artificial) bytecode position -1, since it is invoked upon JVM startup. At bytecode position 16 in the method Demo.sumAreas(Shape[]), there is a polymorphic callsite targeting the methods Composite.area() and Square.area(), respectively, which are represented by two distinct CCT nodes. At that callsite, method Square.area() is dynamically invoked twice, and the execution statistics for these two method executions are stored in the same CCT node, since they occur in the same (callsite-aware) calling-context.

In the example in Figure 1, method Demo.sumAreas(Shape[]) has four basic blocks that are executed one, four, one, and three times, respectively. Note that the default basic block analysis algorithm used by JP2 generates rather large basic blocks [8], as method invocations do not necessarily end basic block. The corresponding control flow graph is related to the factored control flow graph [15].

Figure 3 shows the instrumentation that JP2 inserts into the method Demo.sumAreas(Shape[]). While JP2 operates at the bytecode level, for better readability we illustrate the instrumentation in terms of the corresponding Java code.

For each thread, the current CCT node and the bytecode position of the last (potential) callsite are kept in thread-local variables that are updated upon method entry and completion. These thread-local variables are accessed through the four static methods JP2Runtime.getCurrentNode(), JP2Runtime.setCurrentNode(), JP2Runtime.getBP(), and JP2Runtime.setBP(). For performance reasons, the thread-local variables are implemented as instance fields that have been directly added to the java.lang.Thread class. This is one of the few exceptional cases where JP2 modifies the structure of a classfile.

CCT nodes are represented by instances of type CCTNode. This type offers the methods profileCall and profileBasicBlock. The method profileCall takes a method identifier, the bytecode position of the callsite, and the number of basic blocks in the method as arguments and sets the current node to the callee. String constants stored in the classfile constant pools serve as method identifiers. The method profileCall returns the CCTNode instance rep-

[5] The first instruction in a method has position BP = 1.

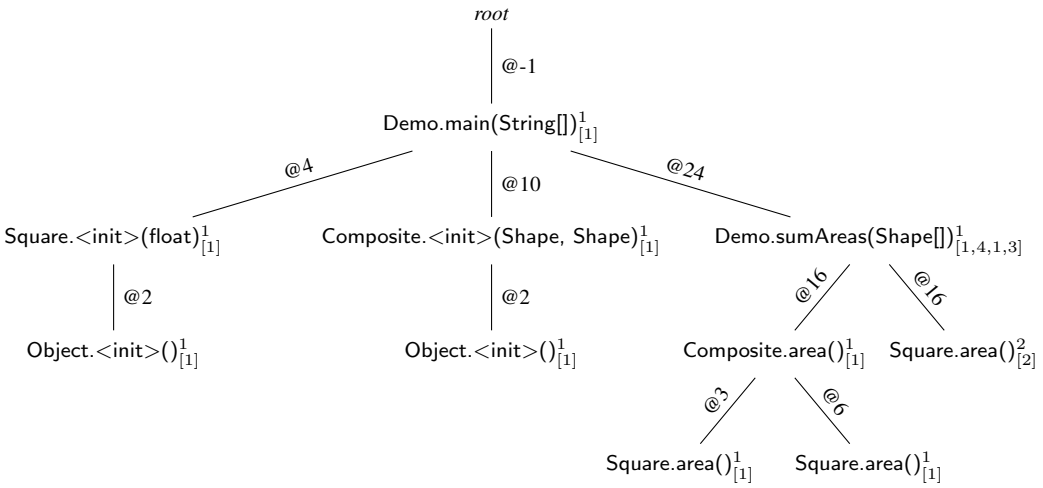

Figure 2. Calling context tree produced when executing Demo.main (cf. Figure 1). Each node contains the number of method invocations (^{m}) and the execution counts of the method's basic blocks ($_{[n_1,n_2,\ldots]}$) in a given calling context. The edges are labelled by the bytecode position at which respective call was made ($@p$).

resenting the corresponding callee node or creates the node if it does not yet exist. An invocation of profileCall is inserted upon each method entry to find (or create) the CCT node representing the current callee. Method profileBasicBlock takes the index of a basic block (which is currently being entered) and increments the corresponding counter in the CCT node.[6] An invocation to profileBasicBlock is inserted at the beginning of each basic block. The implementation details, as far as the CCT's runtime representation is concerned, of CCTNode will be discussed in Section 3.

Before each (potential) callsite, JP2's instrumentation scheme inserts a call to JP2Runtime.setBP() with the position of the bytecode that may trigger a method invocation. Therefore, invocations to JP2Runtime.setBP() are inserted before each invoke bytecode as well as before bytecodes that may trigger class loading or class initialization (e.g., instanceof, new). Hence, dynamic class loading and the execution of class initializers are also represented correctly in the CCT.

JP2 applies its instrumentation scheme for every non-native method, including methods in the Java class library. It thus guarantees completeness of the produced profiles. Instrumenting the Java class library is not trivial, since both the base program being instrumented and the code inserted by JP2 make use of it [7, 33]. JP2 uses polymorphic bytecode instrumentation (PBI) [23] to allow the instrumented and the original versions of the Java class library to coexist. PBI relies on code duplication within method bodies. Depending on the control flow, the corresponding 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 or at the level of the inserted profiling code. Using PBI requires no structural modifications of classfiles.

JP2 is a load-time instrumentation tool relying on a classfile transformer (package java.lang.instrument). Some classes are loaded during JVM bootstrapping before the classfile transformer is registered. JP2 instruments these classes after bootstrapping and redefines them. That is, JP2 relies on the class redefinition (also known as hotswapping) feature provided by modern JVMs, such as by Oracle's HotSpot production VMs.

[6] What is considered a basic block is configurable; in particular, instructions that throw exceptions may or may not end a basic block, depending on the desired level of accuracy [8].

Since native methods have no bytecode representation, JP2 uses native method prefixing (a feature of the JVMTI[7] introduced in JDK 1.6) to wrap each native method with a non-native method that can be instrumented. Details on this approach can be found elsewhere [27].

3. Runtime CCT Representation

Designing a memory-efficient runtime CCT representation is crucial, because the CCT can become very large, literally comprising millions of nodes. For each CCT node, method and callsite identifiers, a method invocation counter, and an array of basic block execution counters need to be stored; to minimize the risk of arithmetic overflow, all counters shall be long values.

Because JP2 has to support profiling of multi-threaded applications, thread-safety of the runtime CCT representation is crucial. There are several ways to achieve thread-safety: maintaining a separate, thread-local CCT for each thread; using synchronization upon CCT access; or using compare-and-swap (CAS) in a thread-safe, non-blocking data structure. The first option may significantly increase the memory footprint for multi-threaded applications. In particular, for applications using thread pools, the CCTs of the worker threads are likely to exhibit similar structure. Furthermore, there may be need for integrating the thread-local CCTs into a shared data structure, for example, upon thread termination. The second option avoids the replication of CCTs and is easy to implement, but may incur high synchronization overhead. JP2 implements the third option, which is more challenging but promises to offer thread-safety in a shared CCT with less overhead than using synchronization.

Figure 4 illustrates JP2's implementation of the class CCTNode. For each node n in the CCT, the nodes representing the callees are kept in a binary search tree rooted at the field n.callees. The fields n.callees.left and n.callees.right refer to the left respectively right branches of this search tree. For each CCTNode instance, the fields left and right therefore represent sibling calling contexts in the CCT. A unique method identifier (including the fully qualified classname, the method name, and the method signature) and the bytecode position of the callsite in the caller method serve

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

```
public final class CCTNode {

    private final String mid; // Method identifier
    private final int bp; // Bytecode position of the callsite
    private final AtomicLongArray bbs; // Execution counters for basic blocks
    private volatile CCTNode left, right; // Siblings in the CCT
    private volatile CCTNode callees; // Children in the CCT
    private volatile long calls; // Number of method calls

    private static final AtomicReferenceFieldUpdater<CCTNode, CCTNode> leftUpdater =
        AtomicReferenceFieldUpdater.newUpdater(CCTNode.class, CCTNode.class, "left");
    private static final AtomicReferenceFieldUpdater<CCTNode, CCTNode> rightUpdater =
        AtomicReferenceFieldUpdater.newUpdater(CCTNode.class, CCTNode.class, "right");
    private static final AtomicReferenceFieldUpdater<CCTNode, CCTNode> calleesUpdater =
        AtomicReferenceFieldUpdater.newUpdater(CCTNode.class, CCTNode.class, "callees");
    private static final AtomicLongFieldUpdater<CCTNode>callsUpdater =
        AtomicLongFieldUpdater.newUpdater(CCTNode.class, "calls");

    private static final CCTNode ROOT = new CCTNode(null, -1, 0);

    private CCTNode(String mid, int bp, int numerOfBBs) {
        this.mid = mid;
        this.bp = bp;
        this.bbs = new AtomicLongArray(numberOfBBs);
        callsUpdater.set(this, 1);
    }

    public void profileBasicBlock(int bbIndex) {
        bbs.incrementAndGet(bbIndex);
    }

    public CCTNode profileCall(String mid, int bp, int numberOfBBs) {
        CCTNode n, alloc;

        // Find or create CCT node. Double-check to reduce risk of creating a node that becomes garbage immediately.
        if ((n = calleesUpdater.get(this)) == null) {
            if (calleesUpdater.compareAndSet(this, null, (alloc = new CCTNode(mid, bp, numberOfBBs)))) return alloc;
            else n = calleesUpdater.get(this);
        }

        int hash_MID = System.identityHashCode(mid);

        while (true) {
            String n_MID;
            if ((n_MID = n.mid) == mid && n.bp == bp) {
                callsUpdater.incrementAndGet(n); return n;
            } else if (hash_MID <= System.identityHashCode(n_MID)) {
                CCTNode nLeft;
                if ((nLeft = leftUpdater.get(n)) == null) { // Find or create CCT node
                    if (leftUpdater.compareAndSet(n, null, (alloc = new CCTNode(mid, bp, numberOfBBs)))) return alloc;
                    else n = leftUpdater.get(n);
                } else {
                    n = nLeft;
                }
            } else {
                CCTNode nRight;
                if ((nRight = rightUpdater.get(n)) == null) { // Find or create CCT node
                    if (rightUpdater.compareAndSet(n, null, (alloc = new CCTNode(mid, bp, numberOfBBs)))) return alloc;
                    else n = rightUpdater.get(n);
                } else {
                    n = nRight;
                }
            }
        }
    }
}
```

Figure 4. CCTNode, a thread-safe, non-blocking data structure using compare-and-swap (CAS).

```
public interface Shape { float area(); }

public class Square implements Shape {
    final float a;

    public Square(float a) {
        super(); // BP = 2
        this.a = a;
    }

    public float area() { return a*a; }
}

public class Composite implements Shape {
    final Shape x, y;

    public Composite(Shape x, Shape y) {
        super(); // BP = 2
        this.x = x; this.y = y;
    }

    public float area() {
        float a1 = x.area(); // BP = 3
        float a2 = y.area(); // BP = 6
        return a1 + a2;
    }
}

public class Demo {
    public static void main(String[] args) {
        Shape s1 = new Square(2); // BP = 4
        Shape s2 = new Composite(s1, s1); // BP = 10
        sumAreas(new Shape[] { s1, s2, s1 }); // BP = 24
    }

    static float sumAreas(Shape[] ss) {
        float sum = 0;
        int i = 0;
        while (true) {
            if (i >= ss.length)
                return sum;
            else
                sum += ss[i++].area(); // BP = 16
        }
    }
}
```

Figure 1. Example base program to be instrumented by JP2 (The abbreviation BP stands for "bytecode position").

as a compound key in the binary search tree. The search algorithm compares keys for identity rather than for equality; this is possible because JP2 stores method identifiers as string constants in the classfiles' constant pools, which are interned by the JVM. Left/right navigation in the binary search tree relies on the keys' identity hash codes rather than on the keys themselves. While the binary search trees are not necessarily balanced, the use of identity hash codes introduces some randomness that reduces the likelihood of degeneration.

The method profileCall searches for the passed compound key (method identifier mid and bytecode position bp) in the binary search tree representing the callees of this node. If a node with the same key is not found, a new CCTNode instance is created and inserted at a leaf node of the binary search tree (i.e., at a node where the left respectively right reference is null). The node insertion relies on CAS; if another thread inserts another node at the same leaf position before the current thread, the CAS operation fails and the search continues.

```
static float sumAreas(Shape[] ss) {
    int callerBP = JP2Runtime.getBP();
    CCTNode caller = JP2Runtime.getCurrentNode();
    CCTNode callee =
        caller.profileCall("sumAreas(Shape[])", callerBP, 4);
    try {
        callee.profileBasicBlock(0);
        float sum = 0;
        int i = 0;
        while (true) {
            callee.profileBasicBlock(1);
            if (i >= ss.length) {
                callee.profileBasicBlock(2);
                return sum;
            } else {
                callee.profileBasicBlock(3);
                JP2Runtime.setBP(16)
                sum += ss[i++].area(); // BP = 16
            }
        }
    } finally {
        JP2Runtime.setCurrentNode(caller);
        JP2Runtime.setBP(callerBP);
    }
}
```

Figure 3. Instrumentation scheme applied by JP2 to the method sumAreas(Shape[]) from Figure 1.

The method invocation counter is an atomic long, and the basic block execution counters are stored in an atomic long array. The size of the array is passed to method profileCall (parameter numberOfBBs), and in turn is passed to the CCTNode constructor. The incrementAndGet operations internally rely on CAS to ensure thread-safety. Method profileBasicBlock takes the number of the basic block entered (bbIndex) and increments the corresponding counter in the atomic long array.

Our data structure allocates only two objects for each (callsite-aware) calling context: an instance of type CCTNode and an atomic long array. Atomic field updaters are used to avoid the allocation of additional AtomicReference and AtomicLong objects. In comparison to alternative representations that use hash tables to keep track of callee nodes, our data structure based on binary search trees results in reduced memory consumption (hash tables should not be completely full to avoid too many collisions, thus wasting some space). Furthermore, thread-safety without using any synchronization would be more difficult to achieve in a data structure based on hash tables.

4. Profile Dumping and Metrics Computation

JP2 offers a flexible plugin mechanism for profile serialization. One or more Dumper classes can be registered at VM startup. At VM shutdown, each dumper is passed the CCT produced by JP2; it can then serialize it in a form suitable for offline analysis.

Currently, JP2 ships with four serialization plugins: two plugins serialize profiles in a text-based format and two serialize profiles as XML-based calling context trees and dynamic call graphs, respectively. Figure 5 shows the XML-based calling context tree produced by JP2 for the Demo class shown by Figure 2. As can be seen, JP2 gathers execution counts for both methods and individual basic blocks. Morever, this information is distinguished by calling context. Each callsite encountered during the program's execution gives rise to a new calling context.

Using an XML output format hereby enables us to rely on off-the-shelf tools for the metric computation. In particular, in this paper we show how one can use XQuery [12] to formulate

```
<callingContextTree>
  <method declaringClass="LDemo;"
          name="main" params="[Ljava/lang/String;" return="V">
    <executionCount>1</executionCount>
    <executedInstructions>30</executedInstructions>
    ...
    <callsite instruction="4">
      <method declaringClass="LSquare;"
              name="&lt;init&gt;" params="F" return="V">
      <executionCount>1</executionCount>
      <executedInstructions>6</executedInstructions>
      <callsite instruction="2">
        ...
      </callsite>
      ...
    </callsite>
    <basicBlock startInstruction="1" endInstruction="30">
      <executionCount>1</executionCount>
    </basicBlock>
  </method>
</callingContextTree>
```

Figure 5. Excerpt from the XML-based calling-context-tree profile (cf. Figure 2) for the Demo class from Figure 1.

metrics as queries with respect to the benchmark's profile. As JP2's profiles do not always contain all the information necessary to compute some metrics, in our case study (cf. Section 5) we make use of another of JP2's features, namely its ability to dump all loaded classes. After converting the classes thus dumped into an XML representation, using a converter shipped with the well-known ASM toolkit,[8] we are now able to access all necessary information by cross-referencing calling context tree and classes using XQuery.

The resource consumption of such an analysis, both in terms of time and space, depends very much on the XQuery processor used. When dealing with large input documents like our XML-based calling-context-tree profiles, whose size may exceed main memory, it thus becomes crucial to use a processor capable of streaming the input document. This of course requires the analysis to be written in such a way as not to require random access; for most analyses we encountered in practice, this is the case.

5. Case Study: Cross-Profiling

To illustrate how to employ JP2 and XQuery to compute custom metrics, we have conducted a case study: cross-profiling [9] for the JOP embedded Java processor [28]. JOP is a Java processor implemented in an FPGA and intended for real-time systems. For such systems, it is important that the worst-case execution time (WCET) of tasks is known. JOP's design thus has a simple timing model to facilitate WCET analysis [30]. In this case study, we will turn the JOP timing model into a query that estimates the number of cycles JOP takes on a selection of benchmarks.

For the case study we use JP2 to profile the six core benchmarks[9] of the JemBench suite [29] for embedded Java. All profiles have been gathered on a desktop Intel Core 2 Duo with 2 GiB of RAM running on the Java HotSpot Server VM with JRE build 1.6.0_24-b07. The JOP processor only acts as a baseline for the profiles' accurary.

Each of these benchmarks was run with a fixed workload using the appropriate harness: fixed.LoopAes, ..., fixed.LoopUdpIp. While JemBench also offers adaptive workloads, using fixed workloads is crucial when working with JP2; the overhead inevitably incurred by profiling must not affect the workload.

For benchmarking in general and for cross-profiling in particular, it is necessary to exclude the code executed during JVM startup and shutdown from the measurements. For this purpose, JP2 offers the option to only profile code in the dynamic extent of the program's main method.[10] While code outside the dynamic extent of the main method still contributes nodes to the CCT, the nodes' execution count is fixed to 0. This property makes it easy to exclude these nodes after the actual profiling with XQuery:

```
declare variable $benchmark-nodes :=
  $cct//cct:method[cct:executionCount > 0];
```

Depending on the benchmark, this excludes between 3,202 and 3,210 nodes from the CCT.

5.1 XQuery for Metric Computation

Using XQuery to estimate the number of cycles needed by JOP to execute each of the benchmarks is conceptually straight-forward:

```
sum(
  for $n in $benchmark-nodes
  for $i in 1 to $n/cct:basicBlock[last()]/@endInstruction
  return jp2:execution-count($n, $i) *
      jop:cycle-count($n, $i))
```

Hereby, the execution count of the i-th instruction can be computed from the CCT alone, provided JP2 was asked to count basic-block executions as well:

```
declare function jp2:execution-count($m, $i) {
  $m/cct:basicBlock[@startInstruction <= $i
      and $i <= @endInstruction]/cct:executionCount
};
```

Computing the number of cycles it takes to execute said instruction on the JOP processor, however, makes it necessary to consult the bytecode instructions of the method in question:

```
declare function jop:cycle-count($m, $i) {
  let $body :=
      asm:instructions($m/jp2:as-asm-method(.))
  let $instruction := $body[$i]
  return typeswitch($instruction)
    case element(NOP) return 1
    case element(ACONST_NULL) return 1
    ...
};
```

While faithful for the vast majority of instructions like nop and aconst_null, the XML representation produced by ASM does introduce a minor imprecision; shorthand instructions like aload_3 are automatically converted to their more general forms like aload. For most uses of ASM this simplification is harmless and often quite convenient. The JOP processor, however, executes the more general instructions slightly slower; thus, the distinction between aload and its shorthands is important. We have addressed this problem by making assumptions about the Java compiler used to compile the code: The optimistic assumption is that whenever the compiler can use the shorthand form of an instruction, it does. The pessimistic assumption is that the compiler always uses the general form of instructions:

```
    ...
    case element(ALOAD) return
```

[8] See `http://asm.ow2.org/asm33/javadoc/user/org/objectweb/asm/xml/Processor`.

[9] JemBench also includes several micro-benchmarks covering, e.g., integer arithmetic; these have not been studied in this case study.

[10] This option should be used whenever the benchmark harness in question does not offer a dedicated callback mechanism suitable for JP2 [27].

```
    if ($instruction/@var <= 3 and
        $optimistic-assumption) then 1
    else 2
  ...
```

A similar problem arises for the ldc instruction, which also exists in a "wide" form , ldc_w; both load a value from a constant pool, but the latter, which accepts a wider index into the constant pool, executes slightly slower on JOP. Unfortunately, the XML representation produced by ASM not only hides the differences between the two forms, but it also completely hides the constant pool. It is thus impossible to reconstruct the actual instruction. We therefore again resort to both an optimistic and a pessimistic assumption. That being said, a study of real-world programs has shown that the shorthands are commonly used [16]; thus, the optimistic assumption is much closer to reality.

But even if assumptions must be made, for most instructions determining the cycle count is straight-forward. For method invocations and returns, however, this proves to be more challenging. The reason is that JOP has to first load the entire target method before control can be transferred. Therefore, the number of cycles it takes JOP to execute a method invocation or return has to include the number of cycles it takes to load said target method. But given the CCT, the target method of a method return is easily obtained:

```
  ...
  case element(RETURN) return 21 +
    max((0, $jop:read-wait-states - 3)) +
    max((0, jop:load-cycles($method/../..) -
        $jop:hidden-load-cycles))
  ...
```

Similar to method returns, method invocations require a new method to be loaded. In the case of dynamically-dispatched method calls, however, the average number of load cycles needs to be computed, as there may be several target methods for a callsite:

```
  ...
  case element(INVOKEVIRTUAL) return 98 +
      2 * $local:read-wait-states +
      max((0, $jop:read-wait-states - 3)) +
      max((0, $jop:read-wait-states - 2)) +
      max((0, jop:avg-target-load-cycles($method, $i) -
        $jop:invoke-hidden-load-cycles))
   ...
```

By querying the CCT, this is easily done, which illustrates how JP2's callsite awareness can be used to good effect in the computation of dynamic metrics:

```
declare function local:avg-target-load-cycles($n, $i) {
  let $targets :=
      $n/cct:callsite[@instruction = $i]/cct:method
  let $execs := sum($targets/cct:executionCount)
  return sum($targets/(jop:load-cycles(.) *
      cct:executionCount div $execs))
};
```

To finish the case study, we only have to fill in the remaining gap, namely the function that computes the number of cycles needed to load a method under one of the two idealized cache regimes, namely one that always hits and one that always misses.

```
declare function jop:load-cycles($method) {
  if ($cache-hits) then 4
  else 6 +
      (ceiling(jop:bytecode-length($method) div 4) + 1) *
        ($jop:read-wait-states + 1)
};
```

Benchmark	JP2 (Pessimistic)		JP2 (Optimistic)		JOP Hardware
	miss	hit	miss	hit	
Kfl	72.28	51.94	66.88	48.49	48.24
Lift	61.08	52.90	56.22	49.26	48.42
Udplp	140.46	113.64	131.44	108.16	108.60
Matrix	75.31	74.83	70.89	70.48	69.42
Queens	262.21	209.51	236.22	197.21	199.32
AES	566.67	513.30	556.19	509.10	454.62

Table 1. Comparison of cross-profiling and hardware execution time (in million clock cycles). Cross-profiling was done both under pessimistic and optimistic assumptions about the Java compiler and for one of two idealized instruction cache regimes (cf Section 5.1).

As the number of cycles it takes JOP to load a method depends on the method's size, the bytecode length must be computed. Again, the XML representation produced by ASM hides this detail, so we approximate the method length both under optimistic and pessimistic assumptions about the compiler's ability to select short instructions.

5.2 Cross-Profiling Accuracy

Table 5.2 shows the execution time of the benchmarks in million clock cycles, both when executed on the physical JOP hardware and when obtained by cross-profiling with JP2. As can be seen, the cross-profiling results that always assume a instruction-cache hit are remarkably close to the number of cycles the JOP hardware takes on the benchmarks, regardless of whether one makes optimistic or pessimistic assumptions about the Java compiler's ability to select the shortest instructions. This is a testimony to the effectiveness of JOP's instruction cache.

What is furthermore interesting is that the effect of a Java compiler not using shorthand instructions is quite noticeable on a hardware implementation like JOP. This effect is even more pronounced when the instruction cache is effectively disabled, i.e., if it always misses, as not using shorthands leads to longer methods and thus to more cycles spend on method invocations and returns.

In general, the accuracy of the simple cross-profiler developed in the section is already quite good. The only benchmark for which the number of cycles taken is significantly overestimated—even assuming a perfect instruction cache—is AES. This is because JOP implements arithmetic operations on 64-bit operands in software; only rough estimates have been made about the cycle counts of these operations for the purpose of this case study. More accurate estimates could be made but are beyond the scope of this tutorial.

Nevertheless, the case study helped to uncover a performance bottleneck in the current JOP implementation: 64-bit multiplication. JOP spends two thirds of its execution time in AES performing multiplications in the runtime's pseudo-random number generator (Random.nextInt()). In future work, we will investigate if this is an issue only on JOP or if other Java processors also spend a considerable time in that single method. In the latter case, this is an indication that the AES benchmark is not well designed and needs to be updated to represent a more varied workload.

6. Performance Evaluation

To assess the performance overhead incurred by the various features of JP2, we have conducted a series of experiments. In all these experiments, we have used the latest release (9.12, nicknamed "Bach") of the DaCapo benchmark suite [11]. Of the suite's 14 benchmarks with default workload size, tomcat has been excluded from our experiments as it always exhibits a StackOverflowError. While this error is caught and does not prevent the benchmark from completing normally, it perturbs our measurements: The CCT

Benchmark	# Nodes			Avg. BB size
	CCT	+ Callsites	+ Natives	
avrora	93,756	187,148	209,391	5.94
batik	29,843	567,288	632,860	7.37
eclipse	13,667,546	20,941,526	21,709,963	5.24
fop	263,951	602,522	667,264	4.84
h2	142,569	328,960	354,491	4.49
jython	10,481,382	21,147,785	23,167,434	5.18
luindex	77,963	235,326	250,349	5.74
lusearch	56,927	91,302	96,443	6.12
pmd	3,364,241	4,362,234	4,602,711	5.05
sunflow	81,264	348,723	385,653	9.38
tradebeans	4,045,407	6,673,170	7,445,191	5.43
tradesoap	4,299,644	7,060,342	7,885,183	5.45
xalan	224,665	387,423	418,536	10.74

Table 2. The number of CCT nodes produced by JP2 for the various DaCapo 9.12 benchmarks [11] for different feature-sets. (When using the basic-block profiling feature, the average size of basic blocks is also given.)

always grows in proportion to the configured stack size. Please note that this error is not caused by JP2; it is a known issue with the benchmark itself (SourceForge issue ID: 2934521). All benchmarks have been run on a Dell PowerEdge M605 with 64 GiB of RAM with two 2.6 GHz AMD six-core Opteron processors (for a total of 12 cores) running on the Java HotSpot Server VM with JRE build 1.6.0_23-b05. 12 GiB of heap have been made available to the benchmarks. All performance measurements are taken for 5 iterations, whereas statistics on the number of CCT nodes are taken for the first iteration only.

Depending on which features of JP2 are used during profiling, the number of nodes in the CCT differs. Table 2 gives details on the CCTs' sizes for the 13 benchmarks chosen for our experiments. As can be seen, callsite awareness significantly increases the number of nodes in the CCT; without this feature subtrees are conflated. Using native-method prefixing in addition increases the CCTs' sizes further, albeit not as much. The average size of the basic blocks is computed from the total number of bytecode instructions executed divided by the numbers of basic blocks executed; in other words, the basic block sizes are weighted by the basic blocks' dynamic execution count.

Table 3 and Figure 6 jointly depict the runtime overhead caused by JP2 with different features enabled. As can be seen, using a callsite-aware JP2 does not incur a significantly higher overhead than using JP2 without callsite awareness, despite the fact that the profiles produced contain many more nodes (cf. Table 2). Although a larger number of nodes needs to be kept in memory, this in itself has little effect on performance, provided that the heap is large enough. (12 GiB proved to be more than sufficient in our experiments.) For a few benchmarks (lusearch and sunflow), whose threads all perform the same set of kernel tasks, performance even improves slightly as an increase in the overall number of nodes reduces the risk of contention when updating a node. Also, the calls to setBP() themselves only incur moderate overhead, making callsite awareness a feature that, on average, incurs little more overhead than JP2 without callsite awareness during the actual profiling. Offline processing of the much larger CCTs is a different matter, though.

In contrast to callsite awareness, native-method awareness does incur significant overhead, even though its effect on the generated CCTs is relatively minor (cf. Table 2); enabling native-method awareness on average adds only 8.8 % more nodes to the results.

The last feature we consider, introduced with the newest version of JP2, is the use of basic-block execution counters. It incurs overhead comparable to native-method awareness. What is interesting here is that benchmarks with a large average basic-block size do not necessarily incur less overhead than benchmarks with a smaller one, the sunflow benchmark being the prime example of this effect. The reason for the lack of strong correlation is that Table 2 reports the average basic-block sizes in bytecode instructions, some of which are much more complex to execute than others. The sunflow benchmark (a raytracer) in particular performs primarily less complex, arithmetic operations. Therefore, the relative overhead of maintaining the basic block execution count is larger for sunflow.

7. Related Work

Much related work exists in the area of profiling, even if one considers profilers for the Java Virtual Machine only.

In 2003, Dufour et al. presented *J [19], a profiler specifically designed for the collection of VM-independent dynamic metrics [18]. However, *J relies on the now obsolete JVMPI [21], usage of which incurs high measurement overhead. JP2 improves not only upon *J in terms of overhead, but also does not rely on JVMPI anymore; it is thus more portable. *J is accompanied by a dedicated trace-analyzer framework. In contrast, the approach advocated in this paper uses an off-the-shelf query language, namely XQuery, to compute dynamic metrics.

To reduce the overhead of profiling, several approaches combine sampling with stack inspection [4, 34]. Such approaches, however, cannot match the accuracy of the profiles produced with JP2. In particular, recent work by Mytkowicz et al. [25] highlights the problems of such an approach. Attempts to further reduce overhead by a technique called "adaptive bursting" [35] have been made. Regardless, by their design all sampling-based approaches cannot produce complete CCTs.

Another approach that trades accuracy for reduced overhead is to compute a probabilistic calling context (PCC) only [14]. Upon each method call, a probabilistically unique value is updated that represents the current calling context. Due to the probabilistic nature of this approach, reconstruction of the CCT from these values is not always possible, although recent work has shown promising results [13]. To the best of our knowledge the PPC approach has so far only been implemented by means of a modified JVM; no portable implementation does exist yet.

Another approach relying on a modified JVM is employed by the NetBeans Profiler.[11] This profiler uses Oracle's JFluid profiling technology [17] for dynamic bytecode instrumentation and code hotswapping. Profiling can thus be enabled and disabled dynamically. Furthermore, it can be restricted to subsets of a program. In contrast, JP2 always instruments the entire program and also produces a CCT reflecting the program's entire execution. However, JP2 also offers an option to easily restrict the actual measurements to only parts of the execution (cf. Section 5).

8. Conclusions

Complete and accurate inter- and intra-procedural profiling of Java applications is important for workload characterization and can support various software engineering tasks such as program optimization, program comprehension, and reverse engineering. In this paper we have presented the new JP2 profiler that can produce such profiles on production JVMs. The CCTs thereby produced by JP2 can distinguish between callsites and provide execution statistics at the level of individual basic blocks of code. JP2 uses bytecode instrumentation techniques and avoids structural modifications of classfiles as much as possible to ensure compatibility with state-of-the-art JVMs. The profiler relies on a special, thread-safe, non-

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

Benchmark	No profiling	CCT (no callsites)		+ Callsites		+ Natives		+ Basic Blocks	
	Time [s]	Time [s]	Overhead	Time [s]	Overhead	Time [s]	Overhead	Time [s]	Overhead
avrora	3.84	33.58	8.74 x	35.32	9.19 x	73.96	19.26 x	115.86	30.17 x
batik	2.39	8.66	3.62 x	7.98	4.54 x	23.39	9.78 x	46.09	19.28 x
eclipse	28.66	135.81	4.73 x	151.16	5.51 x	386.98	13.51 x	743.71	25.94 x
fop	1.15	7.31	6.34 x	7.21	8.31 x	18.14	15.77 x	35.71	31.04 x
h2	7.79	82.86	10.63 x	92.59	11.91 x	291.88	37.46 x	554.25	71.14 x
jython	6.43	64.45	10.02 x	102.92	17.38 x	438.07	68.12 x	607.32	94.44 x
luindex	1.22	11.72	9.60 x	11.95	10.14 x	34.88	28.59 x	72.98	59.81 x
lusearch	1.25	27.55	22.04 x	24.48	19.16 x	34.85	27.88 x	71.21	56.96 x
pmd	2.91	11.29	3.87 x	12.31	4.27 x	22.61	7.76 x	35.81	12.31 x
sunflow	1.67	68.78	41.18 x	54.31	31.06 x	106.49	63.76 x	170.91	102.34 x
tradebeans	7.77	52.21	6.71 x	66.47	8.41 x	128.41	16.52 x	131.69	16.94 x
tradesoap	7.31	37.67	5.15 x	35.55	5.01 x	66.96	9.16 x	120.63	16.52 x
xalan	1.42	16.16	11.38 x	20.47	14.41 x	27.21	19.16 x	33.22	23.39 x
Geo. mean	3.42	29.41	8.55 x	33.05	9.63 x	70.50	20.55 x	117.35	34.21 x

Table 3. Runtime overhead incurred by JP2 with different features for the DaCapo 9.12 benchmarks [11]. The arithmetic mean of 5 runs in the same JVM process (separately started for each benchmark) is shown for each benchmark.

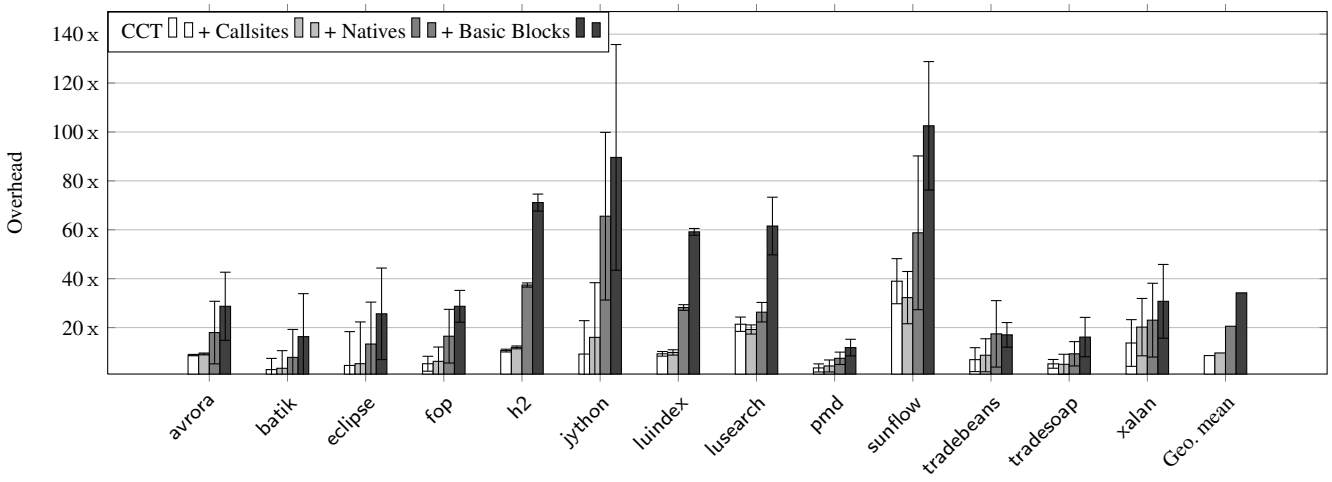

Figure 6. Runtime overhead incurred by JP2 with different features for the DaCapo 9.12 benchmarks [11]. The arithmetic mean ± sample standard deviation of 5 runs is shown for each benchmark.

blocking data structure to represent the profile at runtime, integrating the activities of all threads into a single profile.

JP2 supports custom serialization plugins from whose output various dynamic metrics can be derived. Two of these serialization plugins output the profiles in XML format. This allows researchers to concisely express their metrics in terms of XQuery [12]; metric computation is then done by an off-the-shelf XQuery processor. We demonstrate the convenience and versatility of this workflow with a case-study on cross-profiling for an embedded Java processor.

Depending on the features used, JP2 incurs overhead of a factor between 8.55 x and 34.21 x (geometric mean) for the DaCapo benchmarks. The three significant sources of this overhead are the generation of a callsite-aware CCT, the use of native method prefixing, and intra-procedural profiling at the basic block level; all three features are optional and can be disabled in case the respective information is not needed to compute the desired metrics.

JP2 is Open Source and available to the public under the GNU General Public License. It is currently being employed successfully in a project that aims at characterizing and comparing Java and Scala workloads [31, 32].

Regarding limitations, JP2 currently does not distinguish between different classloaders. That is, if a polymorphic callsite invokes two different target methods with the same name and signature that are defined in distinct classes bearing the same name but defined by distinct classloaders, the two targets will be represented by the same CCT node. Such a situation may yield corrupt profiles, but was not encountered in practice so far. Furthermore, the JVM specification [22] imposes several restrictions on class files that may impair any tool relying on bytecode instrumentation techniques. For instance, method bodies must not exceed 2^{16} bytes in length (indices in exception tables, line number tables, and local variable tables are unsigned 16-bit values). Beyond overcoming the above limitations, future work consists of adding the option to compress recursive cycles in the CCT in order to make the CCT's in-memory representation more space-efficient.

Acknowledgments

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

References

[1] Bowen Alpern, Dick Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, and John J. Barton. Implementing Jalapeño in Java. In *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1997.

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

[4] Matthew Arnold and Peter F. Sweeney. Approximating the calling context tree via sampling. Research Report RC21789, IBM T.J. Watson Research Center, July 2000.

[5] Walter Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, 2005.

[6] Walter Binder. Portable and accurate sampling profiling for Java. *Software: Practice and Experience*, 36(6):615–650, 2006.

[7] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced Java Bytecode Instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ)*, 2007.

[8] Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009.

[9] Walter Binder, Martin Schoeberl, Philippe Moret, and Alex Villazón. Cross-Profiling for Java Processors. *Software: Practice and Experience*, 39(18):1439–1465, 2009.

[10] Walter Binder, Alex Villazón, Martin Schoeberl, and Philippe Moret. Cache-aware cross-profiling for Java processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2008.

[11] 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 21st Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2006.

[12] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, editors. *XQuery 1.0: An XML Query Language*. World Wide Web Consortium, 2nd edition, 2010.

[13] Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[14] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[15] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31. ACM Press, 1999.

[16] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, 2007.

[17] Mikhail Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *Proceedings of the Fourth International Workshop on Software and Performance (WOSP)*, 2004.

[18] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for java. In *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[19] Bruno Dufour, Laurie Hendren, and Clark Verbrugge. *J: A tool for dynamic analysis of Java programs. In *Companion of the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[20] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[21] Sheng Liang and Deepa Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1999.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[23] Philippe Moret, Walter Binder, and Eric Tanter. Polymorphic bytecode instrumentation. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2011.

[24] Philippe Moret, Walter Binder, and Alex Villazón. CCCP: Complete calling context profiling in virtual execution environments. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2009.

[25] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of Java profilers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[26] David Röthlisberger, Marcel Härry, Walter Binder, Philippe Moret, Danilo Ansaloni, Alex Villazón, and Oscar Nierstrasz. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *IEEE Transactions on Software Engineering*, 2011. To appear.

[27] Aibek Sarimbekov, Philippe Moret, Walter Binder, Andreas Sewe, and Mira 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 (BYTECODE)*, 2011.

[28] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1–2):265–286, 2008.

[29] Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, 2010.

[30] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40(6):507–542, 2010.

[31] Andreas 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)*, volume 692 of *CEUR Workshop Proceedings*, 2010.

[32] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java virtual machine. In *Proceedings of the 26th Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2011.

[33] Alex Villazón, Walter Binder, Philippe Moret, and Danilo Ansaloni. Comprehensive Aspect Weaving for Java. *Science of Computer Programming*, 2010.

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

[35] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2006.