SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **46**:1351–1380
Published online 18 December 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2385

Polymorphic bytecode instrumentation

Walter Binder[1,*,†], Philippe Moret[1], Éric Tanter[2] and Danilo Ansaloni[1]

[1]*Faculty of Informatics, Università della Svizzera Italiana (USI), Lugano, Switzerland*
[2]*PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile*

SUMMARY

Bytecode instrumentation is a widely used technique to implement aspect weaving and dynamic analyses in virtual machines such as the Java virtual machine. Aspect weavers and other instrumentations are usually developed independently and combining them often requires significant engineering effort, if at all possible. In this article, we present *polymorphic bytecode instrumentation* (PBI), a simple but effective technique that allows dynamic dispatch amongst several, possibly independent instrumentations. PBI enables complete bytecode coverage, that is, any method with a bytecode representation can be instrumented. We illustrate further benefits of PBI with three case studies. First, we describe how PBI can be used to implement a comprehensive profiler of inter-procedural and intra-procedural control flow. Second, we provide an implementation of execution levels for AspectJ, which avoids infinite regression and unwanted interference between aspects. Third, we present a framework for adaptive dynamic analysis, where the analysis to be performed can be changed at runtime by the user. We assess the overhead introduced by PBI and provide thorough performance evaluations of PBI in all three case studies. We show that pure Java profilers like JP2 can, thanks to PBI, produce accurate execution profiles by covering all code, including the core Java libraries. We then demonstrate that PBI-based execution levels are much faster than control flow pointcuts to avoid interference between aspects and that their efficient integration in a practical aspect language is possible. Finally, we report that PBI enables adaptive dynamic analysis tools that are more reactive to user inputs than existing tools that rely on dynamic aspect-oriented programming with runtime weaving. These experiments position PBI as a widely applicable and practical approach for combining bytecode instrumentations.

Received 9 April 2015; Revised 31 October 2015; Accepted 6 November 2015

KEY WORDS: bytecode instrumentation; dynamic program analysis; profiling; aspect-oriented programming; execution levels; Java virtual machine

1. INTRODUCTION

Virtual machines for safe languages, such as the Java virtual machine (JVM) or .NET, execute platform-independent code – bytecode in the case of the JVM and CLI code in the case of .NET. Many recent programming languages are compiled to virtual machines. For example, Java, Scala, or JRuby programs are compiled to JVM bytecode, and C# programs are compiled to CLI code. Furthermore, there are compilers for recent languages for the partitioned global address space programming model, such as X10 [1] or Fortress [2], which target the JVM.

Instrumentation and manipulation of platform-independent code – subsequently called *bytecode instrumentation* – are key techniques for the implementation of various tools and frameworks. For example, many dynamic program analysis tools, such as profilers and data race detectors, rely on bytecode instrumentation. Many aspect-oriented programming (AOP) languages, like AspectJ [3], are implemented using bytecode instrumentation [4]. Because bytecode instrumentation has become

*Correspondence to: Walter Binder, Faculty of Informatics, Università della Svizzera italiana (USI), Via G. Buffi 13, CH–6900 Lugano, Switzerland.
†E-mail: walter.binder@usi.ch

so central for tool and framework development, modern virtual machines offer dedicated support. For instance, the JVM supports bytecode instrumentation with the JVM tool interface [5] and with the API in the package `java.lang.instrument`. In addition, there are numerous instrumentation libraries for Java bytecode, such as BCEL [6], ASM [7], or Javassist [8], as well as for other languages [9].

Typically, tools relying on bytecode instrumentation are separately developed. Composing several bytecode instrumentations is usually not foreseen and difficult to achieve. However, flexible composition of multiple bytecode instrumentations can enable many interesting applications. For instance, a memory leak detector can analyze a profiler at work. Even if implemented by the same instrumentation tool, interesting compositions like self-application (e.g., a race detector analyzing itself) or *adaptive* dynamic analysis are often out of reach. An adaptive dynamic analysis tool allows the user to select between different analyses for different parts of a program at runtime, thereby avoiding excessive overhead resulting from applying all analyses at the same time for the overall program. JFluid [10] is a good example of an adaptive profiler.

In this article, we present *polymorphic bytecode instrumentation* (PBI), a novel technique that allows several, possibly independent bytecode instrumentations to coexist and to select dynamically which instrumentation takes effect. First, different bytecode instrumentations are applied in isolation to a program class. Afterward, a PBI framework merges the resulting instrumented classes into a single class that holds the code for all applied bytecode instrumentations. For each method, PBI introduces a dispatcher in order to select the desired version of the code at runtime. Because the dispatch logic is customizable, PBI is applicable in a wide range of scenarios.

In addition, PBI also enables bytecode instrumentation of shared libraries, that is, of libraries that are used by the base program as well as by code inserted through instrumentations. A good example of a shared library is the core class library of the considered language, such as the Java class library: in Java, almost any base program invokes methods in the core class library, and many bytecode instrumentations insert code that needs to call some methods in that library. If inserted code invokes already instrumented methods, infinite regression can easily happen. By preventing infinite regression, PBI enables instrumentations with complete bytecode coverage; that is, any method that has a bytecode representation is amenable to bytecode instrumentation, including methods in the core class library. As a special case, aspect weavers implemented with PBI are capable of weaving aspects with complete coverage; this is in contrast with mainstream weavers, such as the standard AspectJ weaver and abc [11].

Polymorphic bytecode instrumentation is a general technique that is applicable to any intermediate language. In this article, we focus on CodeMerger, our PBI framework for Java bytecode. The main contribution of this article is PBI, a general and widely applicable technique that eases the development of instrumentation-based tools, such as software engineering tools for various dynamic program analysis tasks (e.g., profiling, debugging, testing, program comprehension, and reverse engineering) and for implementing advanced AOP frameworks. More concretely, the original, scientific contributions of this article are as follows:

1. We present PBI, a simple and effective technique to dynamically dispatch amongst multiple bytecode instrumentations (Section 2).
2. As a first application, we show how PBI enables instrumentation with complete bytecode coverage without disrupting the virtual machine bootstrapping phase (Section 3).
3. We explain the implementation of CodeMerger (Section 4).
4. We describe three consequent case studies of PBI. First, we use PBI to implement JP2, a comprehensive profiler of inter-procedural and intra-procedural control flow (Section 5). This case study focuses on achieving instrumentation with complete bytecode coverage with the aid of PBI. Second, we use PBI to support *execution levels* [12] in AspectJ (Section 6), thereby enabling black-box composition of dynamic analysis aspects in multiple ways. Third, we leverage PBI to implement *adaptive dynamic analysis* tools, where the dynamic analysis to be performed can be changed at runtime for different parts of the base program (Section 7).
5. We thoroughly evaluate our PBI implementation for Java (Section 8). First, we explore the overhead introduced by PBI dispatch logic and code bloat. Afterwards, we evaluate PBI in the

DOI: 10.1002/spe

three case studies. We report on the performance of JP2 and its ability to cover execution of the core Java libraries. Our evaluation then shows that PBI-based execution levels are much more efficient than equivalent control flow pointcuts to avoid interference between aspects and are generally as efficient as the standard AspectJ weaver when applying analysis aspects on the DaCapo benchmark suite. Finally, we demonstrate that PBI enables adaptive dynamic analysis tools that react more quickly to user inputs than existing tools that rely on dynamic AOP with runtime weaving.

Section 9 discusses prior, ongoing, and related work. Section 10 concludes.

This article extends and refines the work initially presented in [13]. The new contents of this article covers (1) improved deployment options for PBI and an extended discussion of technical details (Sections 4.1 and 4.2); (2) two additional alternative implementations of PBI that solve some serious performance issues (Section 4.4); (3) an additional case study in the profiling domain (Section 5); (4) evaluation results for the new PBI implementations and for the new case study (Sections 8.2 and 8.3); and (5) a new evaluation of PBI-based execution levels, taking both start-up and steady-state performance into account (Section 8.4).

2. POLYMORPHIC BYTECODE INSTRUMENTATION

Many tools make use of bytecode instrumentation to achieve different goals. PBI is a general technique to allow these instrumentations to coexist and to select dynamically which instrumentation takes effect. The name polymorphic stems from the parallel with polymorphic method calls, where the actual code to be executed is chosen dynamically according to some dispatch mechanism. However, as opposed to typical polymorphic calls, the dispatch logic in PBI is not fixed but customizable.

Here, a bytecode instrumentation is considered purely augmentative, meaning it may insert fields and methods,[‡] as well as modify method bodies, but it may not remove any field or method. PBI is applicable to a wide range of bytecode instrumentations,[§] which may be implemented with any instrumentation framework, not necessarily the same. A PBI framework is in charge of integrating these instrumentations, as explained later.

2.1. Polymorphic bytecode instrumentation overview

The PBI enables dynamic dispatch between differently instrumented versions of a method at the granularity of individual method executions. The version of a method to be executed is decided upon method entry. After this selection, it is not possible to switch between differently instrumented parts of a method (e.g., it is not possible to execute a differently instrumented loop body in each loop iteration).

Let us consider $N \geqslant 1$ bytecode instrumentations that are applied to a base program class C_{orig}. Each instrumentation produces an instrumented class, denoted C^{i}_{instr} $(1 \leqslant i \leqslant N)$. These instrumented classes, as well as C_{orig}, are called *class versions*. A PBI framework takes these class versions and merges them into a single class denoted as C_{merged} (Figure 1). There are $N + 1$ class versions considered for merging, where (typically) class version 0 corresponds to C_{orig} and class version i corresponds to bytecode instrumentation i $(1 \leqslant i \leqslant N)$.

At any single point in time, for a given computation step, only one code version is active. Polymorphism comes from *dynamic dispatch* between these versions. Notably, the actual dispatch logic is not fixed by the PBI framework. Rather, it is provided as input (as a `computeCV()` function) in addition to the code versions (Figure 1). The PBI framework uses this dispatch function to insert code that selects the specific version to execute at runtime.

[‡]In this article 'method' stands for 'method or constructor'.

[§]As we will explain in Section 4, PBI imposes some restrictions on bytecode instrumentations. Furthermore, PBI offers a special mechanism for initializing inserted static fields, which is not transparent to bytecode instrumentations.

DOI: 10.1002/spe

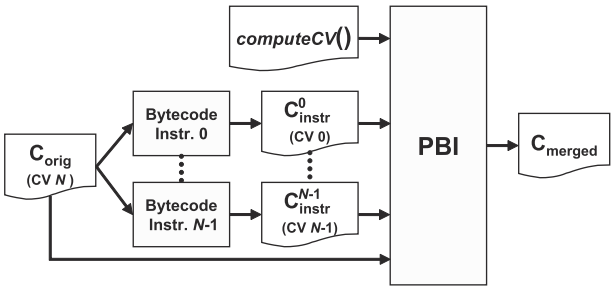

Figure 1. Overview of polymorphic bytecode instrumentation (PBI). First, N different bytecode instrumentations are applied to the original class C_{orig} (class version 0), producing the instrumented classes C^i_{instr} (class versions i, $1 \leqslant i \leqslant N$). The PBI framework merges the class versions into the output class C_{merged}. Each method in C_{merged} has a switch to select the code version to execute; the dispatch logic is defined in the function `computeCV()`.

The merged class C_{merged} generated by the PBI framework has all fields and methods that exist in at least one class version. For methods that have the same signature in different class versions, the corresponding method bodies are merged. We refer to the body of a method defined in class version i as *code version i* of that method. The merged method body starts with the dispatch logic, whose purpose is to jump to the code version to be executed. A PBI framework is free to decide how this jump is realized and where the different code versions effectively reside.

2.2. Dynamic dispatch

Support for dynamic dispatch between code versions is at the heart of PBI. It offers the necessary flexibility to use PBI in a wide range of scenarios, such as complete bytecode coverage (Sections 3 and 5), execution levels for AOP (Section 6), and adaptive dynamic analysis tools (Section 7).

The case studies developed here highlight two different kinds of dispatch logic, based either on (global) state or on control flow. More precisely, both kinds of dispatch logic are typically composed. In the former case, dispatch depends on some value that is accessible to all threads, and so, changes between code versions are global.[¶] This is used, for instance, for adaptive dynamic analysis, where the user globally selects which variant of the analysis to apply. In the latter kind of dispatching, thread-local state is used, thereby allowing different threads to concurrently execute different code versions. This is needed for execution levels, among others.

Also, our case studies show that dispatch logic is typically common to all classes in a program, although some optimizations are applicable to reduce the complexity of dispatch for certain classes [15]. Another case study, described elsewhere [13], explores support for dynamic mixin layers using PBI, illustrating a good scenario for class-specific dispatch.

2.3. Polymorphic bytecode instrumentation for Java: CodeMerger

Our implementation of PBI for Java bytecode is called CodeMerger. The most recent version of CodeMerger uses ASM [7]. The developer of an instrumentation using PBI has to provide each input class version as a pair consisting of the Java class file (represented as a byte array) and the desired version number. The original class C_{orig} is specially marked, allowing CodeMerger to verify whether certain constraints that will be explained later are met. Each input class must have a unique integer version number; the numbering need not necessarily be continuous. While in many cases, it is convenient to assign C_{orig} version number zero, there are also some scenarios where it is convenient to assign a different version number to C_{orig}; an example will be given in Section 6. The function `computeCV()` holding the custom dispatch logic must be provided as a static method in a separate class file. The merged output class is a Java class file.

[¶]In the case of Java, the typical approach is to use fields that are public, static, and volatile, such that their states can be altered asynchronously and the new states become visible to all threads (according to the happens–before relation for volatile writes and reads guaranteed by the Java memory model [14]).

DOI: 10.1002/spe

```
int codeVersion = <computeCV()>; // inlined
switch (codeVersion) {
  case v0: goto CV0; // code version v0 from C_orig
  case v1: goto CV1; // code version v1 from C^1_instr
  ...
  case vN: goto CVN; // code version vN from C^N_instr
  default: goto Error; // code version does not exist
}
CV0: ... // method body from C_orig
CV1: ... // method body from C^1_instr
...
CVN: ... // method body from C^N_instr
Error: throw new IllegalCodeVersionError();
```

Listing 1: Code pattern generated by CodeMerger when merging $N+1$ code versions.

Listing 1 illustrates the generated code pattern for a merged method body. Here, we assume that C_{orig} is assigned version number v_0 and C^i_{instr} is version number v_i. CodeMerger extracts the body of `computeCV()` and inlines it in the beginning of each merged method. The resulting code version is obtained as an integer, and then, a **`switch`** statement dispatches to the appropriate code version. All code versions of a method are simply concatenated in the merged body. Jumping to an unknown code version raises an error at runtime.

If a method exists in more than one class version, PBI requires that its modifiers (i.e., abstract, final, native, static, synchronized, public, protected, private, and strictfp) are the same in each class version. When using PBI with independently developed bytecode instrumentations, it is important to ensure that there is no undesired merging of methods with the same signature. Typically, methods inserted by different bytecode instrumentations need to be renamed (by the developer who implements the PBI-based switching logic) before merging so as to avoid name clashes.

Inserted fields must have different names in each class version, so it may be necessary to rename them to avoid name clashes. Consequently, only the fields in the original class C_{orig} exist in all class versions and are preserved (without any replication) in the merged class C_{merged}. More details about CodeMerger, such as field initialization, are described in Section 4.

3. COMPLETE BYTECODE COVERAGE

Many applications of bytecode instrumentation require complete bytecode coverage in order to function properly. For instance, a profiler needs to be able to track computation occurring in the core language libraries as well as in application code. Binder *et al.* proposed a solution to this issue, albeit in an ad hoc manner [16]. The general technique of PBI subsumes this previous approach.

Instrumentation with complete bytecode coverage implies that every method that has a bytecode representation (i.e., every non-abstract and non-native Java method) must be amenable to bytecode instrumentation, including methods in the Java class library and in dynamically downloaded or generated classes. Full coverage of the Java class library is delicate because of two issues:

1. The instrumentation must not break JVM bootstrapping, for instance, by triggering premature initialization of classes used by inserted code.
2. Code inserted by the instrumentation must not cause infinite regression when invoking methods in the (instrumented) Java class library.

By allowing us to keep both the instrumented method bodies (class versions 1) and the original unmodified method bodies (class versions 0) of the Java class library and dispatching amongst them dynamically, PBI solves both of these issues, as explained hereafter.

DOI: 10.1002/spe

```
public class BootstrapState {
  private static volatile boolean completed = false;

  public static boolean bootstrapCompleted() {
    return completed;
  }

  public static void signalEndOfBootstrap() {
    completed = true;

    // optimization: if supported, redefine this class with a version
    // where bootstrapCompleted() returns the constant true
    ...
  }
}
```

Listing 2: Class `BootstrapState` provides information whether the JVM has completed bootstrapping. Method `signalEndOfBootstrap()` is invoked by the PBI runtime before the base program's main class is initialized.

3.1. Bootstrap with an instrumented Java class library

Many current JVMs are very sensitive to the order in which some core classes in the Java class library (e.g., `java.lang.Object`, `java.lang.String`, or `java.lang.Thread`) are initialized. In such classes, code inserted by a bytecode instrumentation may change the class initialization order when bootstrapping, typically causing a JVM crash.

Because the JVM specification mandates lazy class initialization (JVM Specification, Second Edition, Section 5.5 [17]), inserted code that is not executed during bootstrapping does not change the class initialization order. Hence, we can use PBI in order to execute only the original code version of invoked methods as long as the JVM is bootstrapping. Dispatch is therefore based on a global state that indicates whether the JVM has completed bootstrapping. Class `BootstrapState` (Listing 2) maintains that global state in a static volatile flag. The flag is toggled (by an invocation of `signalEndOfBootstrap()`) before the base program main class is initialized. This can be achieved by calling `signalEndOfBootstrap()` in the `premain(...)` method of a Java agent (package `java.lang.instrument`). Because the flag is volatile, all threads are guaranteed to see the new state of the flag, thanks to the semantics of volatile field access specified by the Java memory model [14]. This state-based dispatch can be simply defined as follows:

```
computeCV() ≡ return BootstrapState.bootstrapCompleted() ? 1 : 0;
```

That is, the instrumented code version is only used after the JVM has completed bootstrapping. Note that our approach will cause initialization of class `BootstrapState` during bootstrapping. However, that class has no static initializer, and reading the Boolean flag during bootstrapping does not trigger any other class initialization. Our approach has been thoroughly tested on many versions of Oracle's HotSpot virtual machines (VM) and IBM's J9 VM.

While in prior work [16] the access to the volatile flag upon each invocation of a method in the Java class library introduced significant extra overhead, some recent state-of-the-art JVMs, such as on Oracle's HotSpot Server VM, enable us to completely eliminate that overhead. If the JVM supports class redefinition (i.e., dynamic class redefinition with the aid of code hotswapping), method `signalEndOfBootstrap()` can replace class `BootstrapState` with a version where `bootstrapCompleted()` returns the constant **`true`**. Thanks to just-in-time compiler optimizations, the overhead due to the check can be completely eliminated.

DOI: 10.1002/spe

```
public class ControlFlow {
  public static boolean inCFlow() {
    Thread t = Thread.currentThread();
    return t.pbi_cflow;
  }

  public static void setCFlow(boolean value) {
    Thread t = Thread.currentThread();
    t.pbi_cflow = value;
  }
}
```

Listing 3: Class `ControlFlow` provides access to boolean, thread-local control flow information.

3.2. *Preventing infinite regression*

If code inserted by an instrumentation invokes some instrumented methods in the Java class library, infinite regression can happen, because the invoked methods would also execute some inserted code. In order to prevent infinite regression, we can keep track of whether a thread is executing code in the *control flow* (i.e., in the dynamic extent) of inserted code, and if so, dispatch to the non-instrumented version of the code. To this end, we need to maintain Boolean control flow information for each thread.

Class `ControlFlow` (Listing 3) provides access to a Boolean, thread-local flag indicating whether the execution is in the dynamic extent of inserted code. We directly insert that flag in class `java.lang.Thread` as the public, Boolean instance field `pbi_cflow`. The control flow-based dispatch is as follows:

```
computeCV() ≡ return ControlFlow.inCFlow() ? 0 : 1;
```

Whenever inserted code may invoke instrumented methods, such as methods in the Java class library, it must first set the thread-local control-flow flag to `true`, and upon completion of the inserted code, it must restore the previous value. That is, in general, the developer of an instrumentation has to use the following code pattern within inserted code that may invoke instrumented methods:

```
boolean old = ControlFlow.inCFlow();
ControlFlow.setCFlow(true);
try {
  ... // inserted code that may invoke instrumented methods
}
finally { ControlFlow.setCFlow(old); }
```

In order to ensure that a bytecode instrumentation properly implements the aforementioned code pattern, the instrumentation may either be manually adapted, or some automated tool may be applied to detect inserted code and to enclose it with the operations that update the control flow information. For example, our aspect weavers MAJOR [18] and HotWave [19] generate the code pattern on code previously woven with the standard AspectJ weaver in a fully automated way.

3.3. *Composite dispatch*

Each of the two issues of complete bytecode coverage, namely, JVM bootstrapping and infinite regression, requires a specific dispatch (respectively state based and flow based). In order to support complete bytecode coverage properly, both dispatch logics must be composed, as follows:

```
computeCV() ≡ return BootstrapState.bootstrapCompleted()
                  && !ControlFlow.inCFlow() ? 1 : 0;
```


DOI: 10.1002/spe

In Section 5, we will present the details of our profiler JP2 that employs PBI with the composite dispatch logic presented here, in order to achieve complete bytecode coverage.

4. CODEMERGER IMPLEMENTATION DETAILS

In this section, we describe our implementation of PBI for Java, CodeMerger. First, we explain the overall process of applying PBI with complete bytecode coverage in Section 4.1. Next, we discuss how CodeMerger handles the initialization of fields inserted by instrumentations; Section 4.2 addresses static fields, whereas Section 4.3 deals with instance fields. Finally, Section 4.4 considers the issue of overlong methods resulting from merging multiple code versions.

4.1. Build-time and load-time instrumentation

CodeMerger can be used for build-time, load-time, and runtime instrumentation. Build-time instrumentation takes place before the instrumented application is started. Load-time instrumentation intercepts class loading events and performs the instrumentation before a class is linked in the JVM. Runtime instrumentation takes place when an application is already running by redefining some previously linked classes. However, class redefinition is severely restricted in some state-of-the-art production JVMs, such as in Oracle's HotSpot VMs. For example, class redefinition may only replace method bodies but must not introduce any new methods or fields. Load-time and runtime instrumentation are supported by the JVM tool interface [5] and by the `java.lang.instrument` API.

The profiling case study presented in Section 5 uses CodeMerger at load time and at runtime. After the JVM has completed bootstrapping, the classes loaded during the bootstrapping phase are redefined with instrumented versions. Afterwards, all other classes are instrumented at load time.

In the other case studies (Sections 6 and 7), we use CodeMerger at build time and at load time. First, the whole Java class library is instrumented at build time. All other classes are instrumented at load time.

4.2. Initialization of static fields

According to the code pattern illustrated in Listing 1, exactly one code version is executed upon each invocation of a merged method. If an instrumentation inserts fields and initializes them to a value different from the default value of the corresponding type, the code pattern in Listing 1 would result in skipping the initialization of some inserted fields depending on the executed code version. On the one hand, skipping initialization of inserted fields can break invariants. On the other hand, requiring instrumentations to leave all inserted fields initialized to their default values would be too restrictive, because many existing bytecode instrumentations initialize inserted fields, in particular static fields. For example, the standard AspectJ weaver inserts static fields and initializes them to hold instances of type `JoinPoint.StaticPart`, holding reflective information of join points [4].

CodeMerger supports the initialization of inserted static fields with the special private static void method `pbi_initClass()`. If a class version needs to initialize inserted static fields, it must do so in its `pbi_initClass()` method, which in turn must be invoked at the end of its static initializer; the `pbi_initClass()` method must not be invoked from any other call site. Upon merging, the `pbi_initClass()` methods and the static initializers in the class versions are treated specially by CodeMerger. First, the `pbi_initClass()` methods are renamed by appending the class version number to the method name. In this way, the bodies of the `pbi_initClass()` methods will not be merged. Second, in each class version, the static initializer is extended to invoke the `pbi_initClass()` methods of all class versions in the end (if there is no static initializer in a class version, it is created). Consequently, after merging of the static initializers, the `pbi_initClass()` methods of all class versions will be executed, independently of the executed code version of the merged static initializer. That is, all inserted static fields will be properly initialized.

DOI: 10.1002/spe

Java requires static final fields to be initialized in the static initializer; it is not allowed to initialize them in another method that is invoked by the static initializer. Hence, it is not possible to initialize static final fields in `pbi_initClass()` methods. Consequently, static fields inserted by an instrumentation must not be declared as final.

In Section 3, we pointed out that during JVM bootstrapping, inserted code – and therefore, also the `pbi_initClass()` method – must not be executed. CodeMerger solves this issue by treating inserted static fields and `pbi_initClass()` methods in the Java class library specially. For each instrumented class C^i_{instr}, the inserted static fields are moved into an extra class in the same package (private visibility is replaced with package visibility), the `pbi_initClass()` method becomes the extra class' static initializer, the invocation of `pbi_initClass()` in the static initializer of C^i_{instr} is removed, and access to the inserted static fields by inserted code in methods in C^i_{instr} is redirected to the static fields in the extra class. Consequently, during JVM bootstrapping, inserted code is not executed and the static fields in the extra classes are therefore not accessed. Because the JVM initializes classes lazily [17], it is guaranteed that the extra classes will not be initialized during JVM bootstrapping. Note that the introduction of extra classes is trivial for build-time instrumentation of the Java class library (the extra classes are simply added to the archive of the instrumented Java class library), whereas in general, it may not be possible to introduce extra classes at load time or runtime, because custom class loaders may not be able to find or may refuse to load the extra classes. However, for load-time and runtime instrumentation after the JVM has completed bootstrapping, CodeMerger does not introduce any extra classes.

Note that PBI is not transparent for instrumentations that insert static fields. In order to use CodeMerger's `pbi_initClass()` feature, existing bytecode instrumentations need to be refactored so as to initialize inserted static fields in the (inserted) `pbi_initClass()` method. In addition, final modifiers on inserted static fields must be removed. As an alternative, post-instrumentation transformations can be performed: for instance, in the case of the AspectJ weaver (Section 6), we apply post-weaving bytecode transformations to move the initialization code for inserted static fields of type `JoinPoint.StaticPart` from the woven static initializer into the `pbi_initClass()` method.

4.3. Initialization of instance fields

CodeMerger does not support initialization of inserted instance fields in the Java class library, as it would be impossible to guarantee that such fields are initialized during JVM bootstrapping. An inserted instance field must be initialized to the default value of the corresponding type. When CodeMerger is applied to AspectJ, this restriction implies that AspectJ's static crosscutting features cannot be fully supported. An inserted instance field can be lazily initialized by inserted code accessing the field, although this incurs extra overhead because of the necessary checks of whether the field has been initialized.

4.4. Dealing with long method bodies

The JVM specification [17] imposes several restrictions on class files, which can impair any application of Java bytecode instrumentation. For instance, method bodies must not exceed 2^{16} bytes (because indices in exception tables, line number tables, and local variable tables are unsigned 16 bit values). While such limitations affect any bytecode instrumentation tool, the merging of code version into a single method body aggravates the problem. This issue can be mitigated by placing code versions in separate private methods when the method size limit is exceeded.

CodeMerger can operate in three different modi. In the first modus, which is the default modus, CodeMerger places multiple code versions into a single method body and does not introduce any new method. In the second modus, CodeMerger puts each code version in a separate private method. In the third modus, which we will call *adaptive modus* in this article, CodeMerger places multiple code versions into a single method body only as long as a specified maximum method size is not exceeded. If the merged method body exceeds that limit, CodeMerger puts the code versions into separate private methods. Note that CodeMerger processes static initializers always in the default

DOI: 10.1002/spe

modus, as static final fields must be initialized within the body of the static initializer (and cannot be initialized in a method invoked by the static initializer).

The default modus has the advantage that it does not introduce any structural modifications of class files; that is, the effects of merging multiple code versions into a single method body are not visible through the reflection API and there are no extra stack frames upon method execution. However, the resulting method bodies can be long, which may have some negative performance impact, for example, if the just-in-time compiler bases decisions on method inlining on the method size (i.e., preventing inlining of long methods). The other two modi produce artifacts (i.e., extra methods) that are visible through the reflection API. However, they help avoid creating methods with very long bodies. In Section 8.2, we will carefully explore the overhead introduced by CodeMerger in each modus.

5. CASE STUDY 1: COMPREHENSIVE PROFILING OF INTER-PROCEDURAL AND INTRA-PROCEDURAL CONTROL FLOW

In this section, we present a profiler, JP2, that relies on PBI to achieve complete bytecode coverage, as explained in Section 3. JP2 profiles both the inter-procedural and the intra-procedural control flow of applications running in any standard JVM.

To capture the inter-procedural control flow of the profiled base program, JP2 instruments each method so as to reify the current calling context for each thread. JP2 maintains a calling context tree (CCT) [20] as a thread-safe data structure shared between all threads in the JVM. Within the thread-local variable `currentCCTNode`, each thread keeps track of its current position in the CCT. Upon method entry, the inserted instrumentation code accesses `currentCCTNode` (which at that moment refers to the CCT node of the caller) and stores the reference in the local variable `callerCCTNode`. Then it looks up the CCT node representing the callee (creating that node if it does not yet exist) and stores the reference in the local variable `calleeCCTNode` as well as in the thread-local variable `currentCCTNode`. Upon (normal and abnormal) method completion, the reference stored in `callerCCTNode` is stored back to `currentCCTNode`.

The profiles generated by JP2 preserve callsite information; that is, if method *m* is invoked from different callsites within the same calling context, the executions of *m* are represented by separate CCT nodes, one for each callsite. Callsite awareness is achieved by storing the bytecode position of a callsite in a dedicated thread-local variable before the call, such that the callee can access and use that information when looking up (respectively creating) its CCT node. Note that it is not sufficient to store the bytecode position only before method invocation bytecodes, but it must be stored also before any bytecode that might trigger class loading or class initialization, as these activities can result in implicit invocations of class-loader methods respectively of static initializers.

JP2 profiles the intra-procedural control flow by incrementing a counter in the beginning of each basic block of code. That is, each CCT node maintains an array of counters, one for each basic block in the body of the method represented by the CCT node.

JP2 uses CodeMerger both at load-time and at runtime. JP2 employs an instrumentation agent written in pure Java that is initialized after JVM bootstrapping, before the first class of the base program to be profiled is loaded. The JP2 agent determines the set of loaded classes and redefines them, replacing them with instrumented versions (i.e., using PBI at runtime). Because instrumentation happens in the same JVM process that runs the instrumented base program, the instrumentation may trigger class loading; these classes need to be instrumented as well. Therefore, the JP2 agent repeatedly determines the set of freshly loaded classes and redefines them, until no further classes are loaded. Then, the agent installs itself to intercept subsequent class loading events. That is, all classes loaded by the execution of the base program will be instrumented at load time.

The instrumentation of JP2 has been carefully designed to avoid structural changes in the classes as much as possible, as such changes would violate current constraints on class redefinition in some production JVMs. JP2 makes only a single structural change to the class `java.lang.Thread`, where it inserts the instance field `pbi_cflow` for control flow-based dispatch, as illustrated in Listing 3 in Section 3.2. This trivial modification of class `java.lang.Thread` is carried out at

DOI: 10.1002/spe

build time (and hence will not interfere with JP2's use of class redefinition). Because JP2 does not insert any static fields, there is no need for using the `pbi_initClass()` feature of CodeMerger.

JP2 has been used for workload characterization at the bytecode level, for the comparison of Java and Scala workloads [21]. In that work, various dynamic metrics (e.g., the number of executed bytecodes, callsite polymorphism, and basic block hotness) are computed by cross-referencing the profiles produced by JP2 with static information from the class files of the profiled base program. The details of the initial design and implementation of JP2 are presented in [22, 23]; later, JP2 was ported to use CodeMerger. JP2 is implemented in ASM [7] and available as open-source software at http://code.google.com/p/jp2/. In Section 8.3, we will explore the overhead of complete bytecode coverage in JP2, enabled by PBI.

6. CASE STUDY 2: EXECUTION LEVELS FOR ASPECTJ

As a second case study for PBI, we explore how the technique makes it possible to implement execution levels [12] for AOP with AspectJ [3].

6.1. Aspects and circularity

An aspect observes the execution of a program through its pointcuts and affects it with its advice. An advice is like a method, and therefore, its execution also produces join points. Similarly, pointcuts as well can produce join points. For instance, in AspectJ, one can use an `if` pointcut designator to specify an arbitrary Java expression that ought to be true for the pointcut to match. The evaluation of this expression is a computation that produces join points. In higher-order aspect languages like AspectScheme [24] and others, all pointcuts and advice are standard functions, whose application and evaluation produce join points as well.

The fact that aspectual computation produces join points raises the crucial issue of the *visibility* of these join points. In all languages, by default, aspectual computation is visible to all aspects – including themselves. This of course opens the door to infinite regression and unwanted interference between aspects. These issues are typically addressed with ad hoc checks (e.g., using **`cflow`** checks in AspectJ) or primitive mechanisms (like AspectScheme's `app/prim`). However, all these approaches eventually fall short for they fail to address the fundamental problem, which is that of *conflating* levels that ought to be kept separate [25].

6.2. Execution levels

In order to address this issue, a program computation is structured in *levels*. Computation happening at level 0 produces join points observable at level 1. Aspects are *deployed* at a particular level, and observe only join points at that level. This means that an aspect deployed at level 1 only observes join points produced by level-0 computation. In turn, the computation of an aspect (i.e., the evaluation of its pointcuts and advice) is reified as join points visible at the level immediately above; therefore, the activity of an aspect standing at level 1 produces join points at level 2.

An aspect that acts *around* a join point can eventually invoke the original computation. For instance, in AspectJ, this is performed by invoking **`proceed`** in the advice body. The original computation ought to run at the same level at which it originated![‖] In order to address this issue, it is important to remember that when several aspects match the same join point, the corresponding advice are chained, such that calling **`proceed`** in advice k triggers advice $k + 1$. Therefore, the semantics of execution levels guarantees that the *last call* to **`proceed`** in a chain of advice triggers the original computation at the lower original level.

This is shown in Figure 2. A call to a `move` method in the program produces a call join point (at level 1), against which a pointcut `pc` is evaluated. The evaluation of `pc` produces join points at

[‖]This issue is precisely why using control flow checks in AspectJ in order to discriminate advice computation is actually flawed. See [12] for more details.

DOI: 10.1002/spe

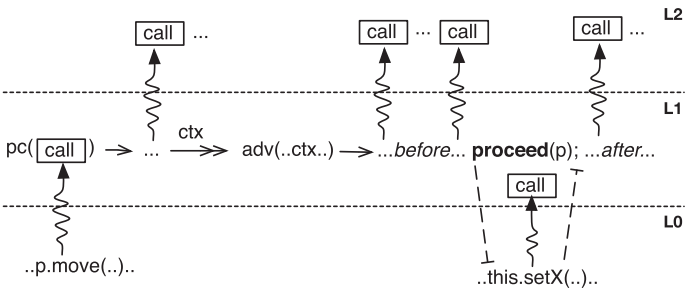

Figure 2. Execution levels in action: pointcut and advice are evaluated at level 1, **`proceed`** goes back to level 0. (from [12]).

level 2. If the pointcut matches, it passes context information `ctx` to the advice. Advice execution produces join points at level 2, except for **`proceed`**: control goes back to level 0 to perform the original computation, then goes back to level 1 for the after part of the advice.

6.3. Execution levels for AspectJ using PBI

Execution levels have been formulated and prototyped in aspect languages with dynamic weaving [12]. In recent work, we have designed an extension of AspectJ with execution levels, tailored to take into account the specificities of AspectJ, like static aspect weaving with partial evaluation of pointcuts [4, 26]. The detailed motivation, design, and applications of this extension are presented elsewhere [15]. Our previous implementation of level switching is carried out in an ad hoc manner; here, we describe how PBI can be used for that sake. Section 8 also provides a much more detailed performance evaluation of the implementation.

Semantically, the execution of a method produces join points. These join points may be seen by pointcuts that may match them; if so, the corresponding pieces of advice are triggered. In aspect languages that perform weaving statically, join point production is partially evaluated [26]: based on the static properties of code, it is determined whether or not a given expression can produce a join point that will be matched at runtime [4]. If so, such a join point *shadow* is transformed so as to invoke advice appropriately. If it can be statically determined that the pointcut however never matches join points corresponding to the shadow, then no transformation happens. The matching of the pointcut may also depend on runtime information not available at compile time; in that case, the shadow is woven together with a *residue*, that is, a conditional expression that guards the invocation of the advice.

```
public class ExecutionLevel {
  public static int currentLevel() {
    Thread t = Thread.currentThread();
    return t.pbi_level;
  }

  public static void up() {
    Thread t = Thread.currentThread();
     ++t.pbi_level;
  }

  public static void down() {
    Thread t = Thread.currentThread();
    --t.pbi_level;
  }
}
```

Listing 4. Class `ExecutionLevel` provides access to the execution level of the current thread.

DOI: 10.1002/spe

```
// shifting up
ExecutionLevel.up();
try {
   ... // advice and pointcut residues
}
finally { ExecutionLevel.down(); }

// shifting down
ExecutionLevel.down();
try {
   ... // original computation
}
finally { ExecutionLevel.up(); }
```

Listing 5. Patterns of level-shifting.

With execution levels, the join points produced by the execution of a method vary. If base program code, running at level 0, invokes a method, it produces join points at level 1, that may be matched by aspects deployed at that level. If an aspect deployed at level n calls this same method, then it produces join points at level $n + 1$, visible only for aspects deployed at level $n + 1$. We use PBI to check the execution level upon method entry and dispatch appropriately to a particular code version. More precisely, there is one code version per execution level, and each code version corresponds to the code with the instrumented shadows of the aspects deployed at the level directly above it. For instance, execution at level 0 uses code version 0, which is the result of weaving aspects deployed at level 1. Execution at level N (the highest level in the configuration) uses code version N, which is set to be the original, non-instrumented code version. A code version is obtained by invoking the standard AspectJ weaver with the aspects deployed at a given level.

In order to track execution levels, we define a class `ExecutionLevel` that provides access to a thread-local variable that indicates at which level the current thread is running (Listing 4). For that, we insert an integer instance field `pbi\_level` in class `java.lang.Thread` to keep track of the current level. Method `currentLevel()` returns the current thread's level, whereas methods `up()` and `down()` increment respectively decrement it.

Level shifting is carried out upwards for the dynamic extent of both advice and pointcut residues, following the top pattern of Listing 5. A level shift downwards occurs for around advice, when the original computation is finally called with **`proceed`**, following a similar pattern (Listing 5, bottom).

Our PBI-based aspect weaver, MAJOR2, uses the unmodified standard AspectJ weaver and post-processes its output to automatically insert the above pattern in each advice method and in each method corresponding to compiled **`if`** pointcuts.

The dispatch logic given to the PBI framework simply consults the current execution level and dispatches to the corresponding version. Finally, because execution levels generalize the solution we presented in the previous section to avoid infinite regression, we only need to combine the levels check with the JVM bootstrap check in order to obtain execution levels for aspects with complete bytecode coverage:

```
computeCV() ≡ return BootstrapState.bootstrapCompleted() ?
                    ExecutionLevel.currentLevel() : N;
```

The evaluation in Section 8.4 uses two different deployment configurations with two aspects in order to assess the performance of the PBI-based implementation. Additional deployment configurations are discussed in [15].

DOI: 10.1002/spe

7. CASE STUDY 3: ADAPTIVE DYNAMIC ANALYSIS

Adaptive dynamic program analysis allows the user to choose or change the dynamic analysis to be performed at runtime. For example, in adaptive profiling, the profiler code is adapted at runtime based on user choices, in order to restrict profiling to only part of an executing application or to enable and disable the collection of certain dynamic metrics. Adaptive profiling helps reduce profiling overhead, as only data of current interest are gathered.

A good example of an adaptive profiler is JFluid [10], which has been integrated in the NetBeans Profiler [27]. JFluid measures execution time for selected methods and generates a CCT (like JP2, Section 5) to help analyze the contributions of direct and indirect callees to the execution time of selected methods. JFluid is an *adaptive* profiler: when the user selects different methods for profiling at runtime, JFluid adapts the profiling code accordingly, using the class redefinition mechanism of the JVM.

Runtime instrumentation and class redefinition can be very expensive, in particular if many classes are to be instrumented and if the instrumentation is specified in a high-level programming model, such as AOP, which requires more complex tool support (e.g., in the case of AOP, an aspect weaver is used). For example, with the dynamic AOP framework HotWave [19], which relies on runtime aspect weaving and on class redefinition, weaving an aspect into all modifiable classes at runtime may take up to 60 s on a recent machine (Section 8.5).

If the set of bytecode instrumentations that may be needed is known in advance, it is not necessary to resort to expensive class redefinition techniques. Instead, we can use PBI to apply all the instrumentations and decide at runtime which code version to execute. For example, Villazón *et al.* present an adaptive profiler built with HotWave that may switch between two different instrumentations (implemented as aspects) at runtime [19]. The default instrumentation generates a plain CCT, whereas a second instrumentation additionally stores various dynamic metrics in the CCT nodes. The second instrumentation introduces much higher overhead and therefore is applied at runtime only to the classes for which the user desires detailed dynamic metrics. Instead of applying the two different instrumentations (possibly repeatedly) at runtime by redefining the affected classes, PBI allows us to merge the code versions for both instrumentations and to switch between them at runtime.

Figure 3 illustrates a case of adaptive dynamic analysis with PBI: the user defines, and changes dynamically, the *scope* of the profiling; profiling data are then passed to a profiling agent that renders it. Implementation-wise, all methods have two code versions and start with a dispatch that triggers the appropriate version, based on the current scope definition.

In the general case, `computeCV()` dispatches between N different instrumentations based on asynchronous user choices. These user choices may be at the level of classes or packages.

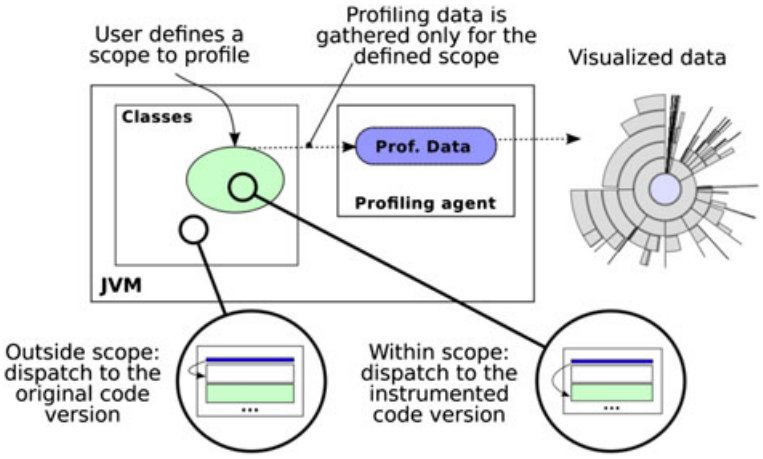

Figure 3. Adaptive dynamic analysis with polymorphic bytecode instrumentation.

DOI: 10.1002/spe

Depending on the granularity at which the user can switch between instrumentations, we assume there is some state (i.e., a public static volatile field) for each class or package indicating the code version to be executed. The effect of a state change is similar to class redefinition in current JVMs: all subsequently invoked methods will read the new state and execute the corresponding code version, whereas methods that already executed the dispatch logic before the state change are not affected. The `computeCV()` function as follows is a template where the meta-variable `selectedCodeVersion` refers to the corresponding volatile field to be read:

```
computeCV() ≡ return BootstrapState.bootstrapCompleted() &&
                    !ControlFlow.inCFlow() ? selectedCodeVersion : 0;
```

This dispatch logic uses the bootstrapping state and the control flow information in the same way as explained in Section 3, in order to enable instrumentation of the Java class library. Code version 0 corresponds to the original method bodies in C_{orig}. Note that for methods in the base program, `computeCV()` can be optimized as follows, assuming that the inserted code never invokes any method of the base program:

```
computeCV() ≡ return selectedCodeVersion;
```

As mentioned in Section 3, reading a volatile variable upon each method entry may introduce significant overhead. If the user rarely changes his choice of the code version to be executed (by writing to the meta-variable `selectedCodeVersion`), redefining the class that holds the volatile variable (as explained in Section 3) helps reduce the overhead of reading the volatile variable in state-of-the-art JVMs. Because of the de-optimization and re-optimization caused by class redefinition, changing the volatile variable does introduce some temporary overhead. However, compared with a solution based on runtime instrumentation and on possibly redefining (potentially) all previously loaded classes, this approach only redefines a single, trivial class. Furthermore, this approach supports the *atomic* change of a set of instances of the meta-variable `selectedCodeVersion` (e.g., atomically changing the instrumentation for a set of classes or for a set of packages).

8. EVALUATION

In this section, we thoroughly evaluate CodeMerger, our PBI implementation, in different scenarios. Section 8.1 summarizes our measurement environment and evaluation settings. In Section 8.2, we explore the performance impact of code duplication introduced by PBI, considering the three different modi supported by CodeMerger (Section 4.4), that is, placing all code versions into a single method body, generating a separate private method for each code version or adapting to the method size. In Section 8.3, we investigate the performance overhead of complete bytecode coverage in the profiling case study (Section 5). In Section 8.4, we evaluate MAJOR2, our PBI-based implementation of execution levels for AspectJ (Section 6), considering both start-up and steady-state performance. Finally, Section 8.5 presents our evaluation of an adaptive dynamic program analysis tool (Section 7) and compares it with an alternative implementation that relies on dynamic class redefinition.

8.1. Measurement environment

Our measurement machine is a quad-core machine (Dell Optiplex 760, 1 quad-core Intel CPU, 3.0 GHz, 8 GB RAM) running Fedora 13 and the Oracle JDK 1.6.0_18 Hotspot Server VM (64 bit version with 4 GB maximum heap size). In Section 8.2, we additionally use the HotSpot VM in interpreted mode, to study the base overhead introduced by PBI in a JVM both with just-in-time compilation and with interpretation.

We use the DaCapo benchmarks (dacapo-2006-10-MR2) [28] with the default workload size. For some evaluations, we also show the geometric mean of the measurements for all DaCapo benchmarks.

DOI: 10.1002/spe

8.2. Overhead of polymorphic bytecode instrumentation dynamic dispatch

In this subsection, we evaluate the base overhead introduced by PBI when two identical code versions (without any instrumentation) are merged. We consider all three modi supported by CodeMerger, that is, (1) merging code versions within method bodies, (2) introducing a private method for each code version, and (3) the adaptive modus that creates extra methods only if the size of the merged code versions (and the inserted dispatch logic) would exceed a given limit. We use the default value of the JVM parameter `-XX:FreqInlineSize` as limit; this parameter indicates the maximum size of methods that are inlined when executed frequently. The rationale of this choice is to avoid extra methods as long as it does not prevent inlining of frequently executed methods. Regarding dynamic dispatch, we consider three different types of `computeCV()` function; the first reads a thread-local variable; the second reads a static volatile field, and the third reads a static final field. The thread-local variable, respectively volatile or final field, always contains the value 0; that is, always the same code version is executed for each invoked method.

Figure 4 shows that in interpreted mode and with a dispatch logic based on a thread-local variable, extending the size of method bodies is always faster than using extra methods. The reason is that the interpreter does not perform any inlining and the extra method calls always introduce some overhead. The adaptive strategy performs almost as good as always merging code versions into the same method body, because the majority of frequently executed method bodies are still smaller than the limit of the adaptive strategy. Therefore, extra methods are not created in most cases.

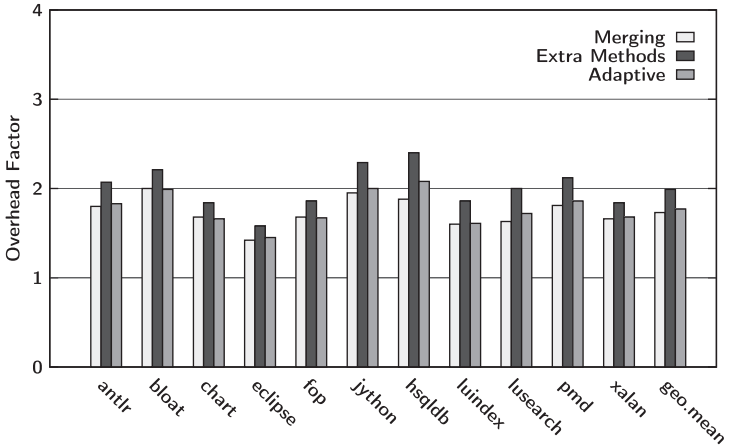

Figure 4. Overhead of polymorphic bytecode instrumentation with two identical code versions; interpreter and thread-local dispatch.

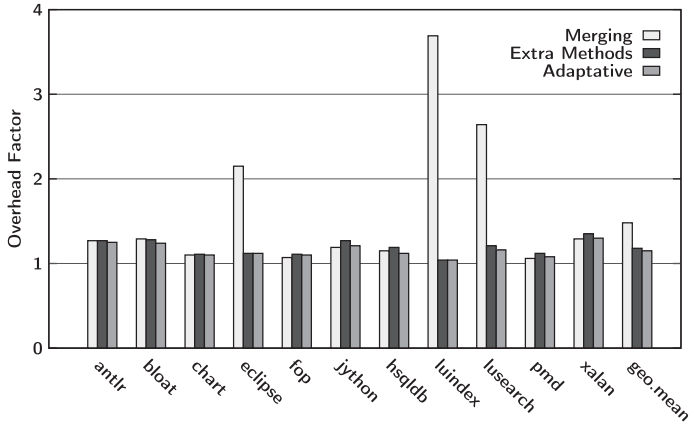

Figure 5. Overhead of polymorphic bytecode instrumentation with two identical code versions; HotSpot server compiler and thread-local dispatch.

DOI: 10.1002/spe

Using the same dispatch logic, Figure 5 illustrates PBI overhead when the JVM uses just-in-time compilation. For most benchmarks, the overhead is rather low in all three modi, between 5% and 20%. However, for three benchmarks (i.e., eclipse, luindex, and lusearch), always merging code versions into the same method body introduces surprisingly high overhead between factor two and factor four. The reason for this excessive overhead is that some frequently executed methods cannot be inlined anymore because their size exceeds the limit imposed by the JVM. Always generating extra methods avoids this problem but introduces slightly more overhead for some other benchmarks (e.g., jython, hsqldb, pmd, and xalan); in comparison with interpreted mode (Figure 4), the overhead because of extra method calls is less pronounced, as the just-in-time compiler is able to inline many extra methods. The adaptive strategy always outperforms the use of extra methods for all code versions; it succeeds in combining the benefits of the other two strategies.

Figure 6 shows that in the interpreted mode of Oracle's HotSpot VM, accessing a thread-local variable is much more expensive than accessing a static volatile field or a static final field. Because there is no just-in-time compilation, there is no evident performance difference between accessing a volatile respectively final field, as access to final fields is not optimized.

In contrast, Figure 7 illustrates that with just-in-time compilation (i.e., HotSpot server compiler); access to a volatile field is more expensive than access to a thread-local variable, which in turn is slightly more expensive than access to a final field.

In summary, we conclude that the base overhead introduced by CodeMerger is small on a modern JVM with an optimizing just-in-time compiler, whereas it may reach a factor of 2 when using

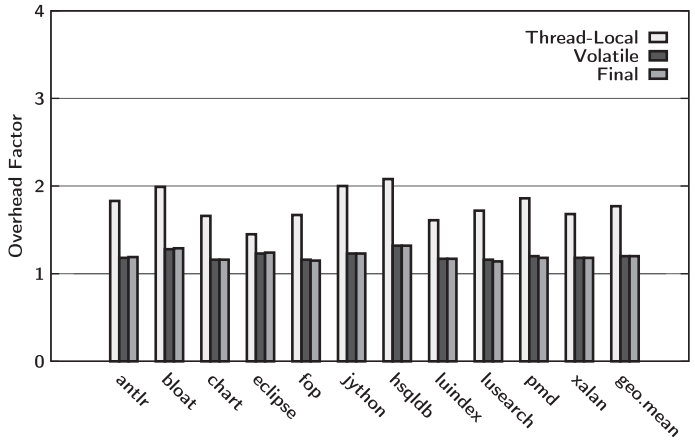

Figure 6. Overhead of polymorphic bytecode instrumentation for different `computeCV()` functions; interpreter and adaptive strategy.

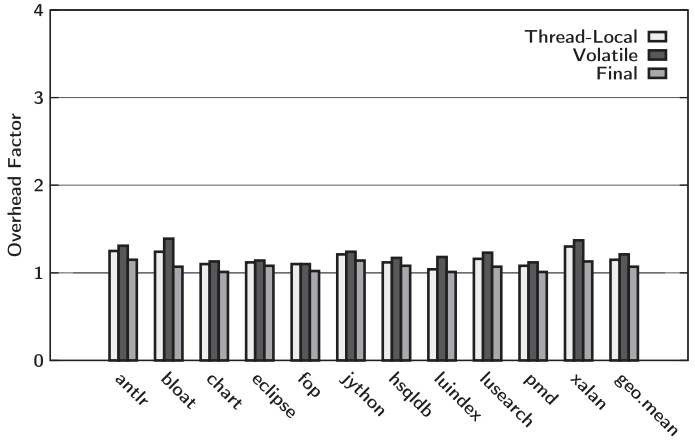

Figure 7. Overhead of polymorphic bytecode instrumentation for different `computeCV()` functions; HotSpot server compiler and adaptive strategy.

DOI: 10.1002/spe

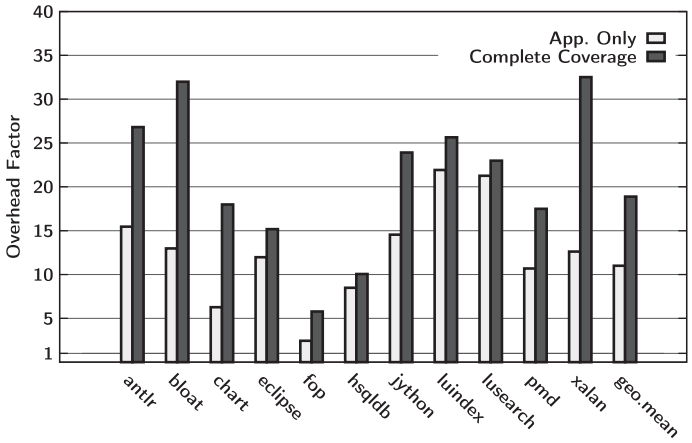

Figure 8. Overhead of JP2 with and without complete bytecode coverage.

Table I. Number of executed basic blocks of code and methods depending on the coverage.

	Executed basic blocks			Method invocations		
	Application only	Complete coverage	Increase factor	Application only	Complete coverage	Increase factor
antlr	3.63E+08	6.13E+08	1.69	9.37E+07	1.63E+08	1.74
bloat	6.82E+08	2.66E+09	3.90	4.92E+08	1.05E+09	2.12
chart	4.55E+08	1.54E+09	3.38	1.60E+08	3.83E+08	2.39
eclipse	5.07E+09	6.83E+09	1.35	5.76E+08	8.99E+08	1.56
fop	4.40E+07	1.35E+08	3.07	1.45E+07	4.20E+07	2.90
hsqldb	5.33E+08	5.83E+08	1.09	1.83E+08	2.18E+08	1.19
jython	7.62E+08	1.86E+09	2.44	3.17E+08	6.81E+08	2.15
luindex	1.51E+09	1.77E+09	1.17	3.56E+08	4.25E+08	1.20
lusearch	1.15E+09	1.60E+09	1.39	3.36E+08	4.52E+08	1.35
pmd	6.45E+08	9.92E+08	1.54	2.90E+08	4.62E+08	1.59
xalan	1.06E+09	2.25E+09	2.13	2.78E+08	6.31E+08	2.27
geo.mean			1.92			1.79

an interpreter. The adaptive mode of CodeMerger achieves consistently good results for all benchmarks, both when using just-in-time compilation and interpretation. In contrast, the default modus may result in surprisingly high overhead in certain situation when just-in-time compilation is used.

8.3. JP2

We now evaluate the impact of complete bytecode coverage in our profiling case study (Section 5). Figure 8 reports the profiling overhead introduced by JP2 in two different settings: (1) instrumenting only the base program classes and (2) instrumenting all classes; only the second setting requires the use of PBI to achieve complete bytecode coverage.

In both settings, the profiling overhead is high, ranging from a factor of 2 to a factor of 33. The high overhead is not surprising, as JP2 performs a heavyweight instrumentation, including callsite-aware calling-context profiling and intra-procedural profiling at the basic block level. Furthermore, the profile data structure is shared between all threads, incurring additional overhead for thread-safety, particularly for multi-threaded benchmarks such as lusearch and xalan. For a detailed exploration of the different sources of overhead,[**] we refer to [22]. Here, we are only interested in the performance impact of complete bytecode coverage as enabled by PBI.

[**] Note that the measurements reported in this article are not directly comparable with the measurements published in [22] because of different measurement environments.

DOI: 10.1002/spe

Figure 8 shows that on average (geometric mean for the DaCapo benchmarks), JP2 with complete bytecode coverage introduces almost twice as much overhead as JP2 instrumenting only classes of the base programs. However, the bigger part of the extra overhead of instrumenting the Java class library does not stem from code duplication by PBI (as we confirmed in Section 8.2) but from the fact that much more code is instrumented, as explained later.

Table I summarizes the number of executed basic blocks and the number of method invocations, both for JP2 with complete bytecode coverage and for JP2 instrumenting only the classes of the base program. Some benchmarks (i.e., bloat, chart, fop, jython, and xalan) trigger the biggest part of events in the Java class library (both in terms of executed basic blocks and method invocations). It is not surprising that for these benchmarks, complete bytecode coverage can introduce more than 2.5 times the overhead of covering only base program code. The correlation coefficient between the difference in overhead (complete bytecode coverage versus covering only base program classes) and the difference in the number of executed basic blocks (resp. in the number of method invocations) is 0.8523 (resp. 0.8531). These results confirm that the extra overhead for complete bytecode coverage indeed stems from the larger amount of data collected.

This study of coverage with the Dacapo benchmarks shows that computation within the Java class library is in fact a big part of the overall activity of Java programs (almost twice as many basic blocks are executed and almost twice as many methods are invoked). This result is especially important in that it demonstrates that proper profilers cannot ignore the core libraries: complete bytecode coverage is crucial, and PBI is an efficient technique to achieve it when implementing profilers in pure Java.

8.4. Execution levels

In the following evaluation, we consider the performance impact of execution levels in AOP. First, we describe the aspects used in our evaluation and the deployment scenarios in Section 8.4.1. Second, we explore the overhead of execution levels in comparison with the standard AspectJ load-time weaver in Section 8.4.2. Third, we investigate the overhead when weaving with complete bytecode coverage in two different deployment scenarios in Section 8.4.3.

8.4.1. Profiling aspects and scenarios. Our evaluation is carried out with two bytecode instrumentations for dynamic program analysis implemented as aspects, the object allocation profiler `ProfAllocs` and the method call profiler `ProfCalls` (Listing 6). The allocation profiler collects the number of object allocations for each class, and the method call profiler collects the

```
public aspect ProfAllocs {
  after() returning(Object o) : call(*.new(..)) && ScopeProf.scope() {
    profileAllocation(o.getClass()); // not shown here
  }
  ...
}

public aspect ProfCalls {
  pointcut allExecs() : (execution(* *(..)) || execution(*.new(..)));
  before() : allExecs() && ScopeProf.scope() {
    profileCall(thisJoinPointStaticPart); // not shown here
  }
  ...
}

public aspect ScopeProf {
  pointcut aspects() : within(ProfAllocs) || within(ProfCalls);
  pointcut scope() : !aspects() && !cflow(aspects());
}
```

Listing 6: Simplified aspects for object allocation and method call profiling.

DOI: 10.1002/spe

number of method calls for each method. Both profilers maintain a thread-safe mapping from identifiers to atomic longs (methods `profileAllocation(...)` and `profileCall(...)`, which are not shown in the listing). For `ProfAllocs`, the identifiers are the classes represented by `java.lang.Class` instances, and for `ProfCalls`, the method identifiers are represented by `JoinPoint.StaticPart` instances (from AspectJ). We use non-blocking data structures from the `java.util.concurrent` package, concretely `ConcurrentHashMap` and `AtomicLong`. We discuss the scoping pointcuts defined in `ScopeProf` in Section 8.4.2.

We weave the `ProfAllocs` and `ProfCalls` aspects in the DaCapo benchmarks that serve as base programs. We use our new PBI-based re-implementation of MAJOR2 [15] that relies on CodeMerger, which provides support for execution levels and complete bytecode coverage. Aspects are woven with AspectJ 1.6.5 (MAJOR2 is also based on AspectJ).

We considered two scenarios for this evaluation:

1. `ProfAllocs` and `ProfCalls` are applied to the base program (i.e., both aspects are deployed at level 1).
2. `ProfCalls` is applied to the base program (i.e., deployed at level 1), and `ProfAllocs` is applied to `ProfCalls` (i.e., deployed at level 2), thus profiling object allocation in `ProfCalls`.

8.4.2. Comparison with AspectJ. Our first evaluation compares the performance of code woven with AspectJ's load-time weaver (henceforth called ajc-ltw) versus MAJOR2. As explained in Section 6, MAJOR2 relies on PBI to implement control flow-based dispatch based on a thread-local field. In this comparison, we use scenario 1, as this is the only composition scenario that AspectJ can handle. We limit the coverage of MAJOR2 to application classes in order to have comparable settings. For each benchmark of the DaCapo suite, we report the first run for start-up performance and we take the median of 15 runs executed in the same JVM process to evaluate steady-state performance. We also compute the geometric mean for all benchmarks except bloat and eclipse.[††]

The aspects in Listing 3.2 use a `scope()` pointcut in order to avoid infinite regression caused by their own computation, as well as to avoid seeing join points produced by the other aspect. Using control flow checks for achieving this is the most robust pattern, as it ensures that all join points in the dynamic extent of aspect executions are ignored. This pattern is well known [29] and is used in many aspect implementations [30]. In our MAJOR2 implementation, these pointcuts are not needed at all because execution levels already address the issues of infinite regression and mutual visibility. Our benchmarks therefore enable us to compare the cost of these typical control flow checks and of our implementation of execution levels.

For further comparison, we also benchmark an optimized version of the aspects with ajc-ltw, where we skip the control flow checks and just leave the lexical `aspects()` condition. This happens to be safe in this particular case, because all potential sources of regression and interference are situated lexically in the aspect definitions, no shared libraries are woven, and there are no callbacks from the aspects to the base code.

Tables II and III show the measured execution times and overhead factors respectively for steady-state and start-up performance. Figures 9 and 10 visualize the overhead factors reported in the tables.

Considering steady-state performance, MAJOR2 introduces significantly less overhead than ajc-ltw (factor 6.28 for MAJOR2 versus factor 11.79 for ajc-ltw, on average). This confirms that PBI-based execution level dispatch can be much more efficient than the use of control flow pointcuts for avoiding infinite regression and aspect interferences. As expected, the optimized ajc-ltw performs better than MAJOR2, as it does not incur the PBI overhead. However, the difference is relatively small (overhead factor 5.80 for the optimized ajc-ltw versus factor 6.28 for MAJOR2, on average). Recall that the optimization is fragile and not generally applicable.

[††]We exclude bloat because it fails with ajc-ltw (in contrast to MAJOR2). We exclude eclipse because ajc-ltw fails to weave a large number of classes because of dependencies (ajc-ltw depends on classes that are also used by the eclipse benchmark); such a problem does not exist with MAJOR2, which makes proper use of class-loader namespaces.

DOI: 10.1002/spe

Table II. *Steady-state* overhead comparison between AspectJ and MAJOR2 in scenario 1; median of 15 runs. Aspects are woven only into application classes.

	Orig.	ajc-ltw		ajc-ltw (opt.)		MAJOR2	
	[ms]	[ms]	Ovh.	[ms]	Ovh.	[ms]	Ovh.
antlr	796	13,002	16.33	5252	6.60	5510	6.92
chart	2863	26,371	9.21	10,229	3.57	10,896	3.81
fop	1106	3064	2.77	1901	1.72	1921	1.74
hsqldb	2535	28,201	11.12	11,529	4.55	11,729	4.63
jython	2393	40,584	16.96	19,908	8.32	22,330	9.33
luindex	3422	53,504	15.64	19,719	5.76	28,581	8.35
lusearch	1333	22,368	16.78	14,784	11.09	14,678	11.01
pmd	2396	37,977	15.85	22,895	9.56	23,387	9.76
xalan	1163	15,633	13.44	9246	7.95	9944	8.55
geo.mean			11.79		5.80		6.28

Table III. *Start-up* overhead comparison between AspectJ and MAJOR2 in scenario 1; first run only. Aspects are woven only into application classes.

	Orig.	ajc-ltw		ajc-ltw (opt.)		MAJOR2	
	[ms]	[ms]	Ovh.	[ms]	Ovh.	[ms]	Ovh.
antlr	1839	15,839	8.61	7550	4.11	11,181	6.08
chart	5277	30,540	5.79	14,515	2.75	23,470	4.45
fop	1952	6612	3.39	5314	2.72	19,221	9.85
hsqldb	3683	29,293	7.95	14,334	3.89	18,098	4.91
jython	6886	47,142	6.85	25,189	3.66	30,603	4.44
luindex	4448	55,745	12.53	21,017	4.73	32,067	7.21
lusearch	4374	23,933	5.47	15,274	3.49	18,525	4.24
pmd	4783	41,815	8.74	27,132	5.67	38,103	7.97
xalan	4475	19,567	4.37	11,832	2.64	14,754	3.30
geo.mean			6.61		3.63		5.51

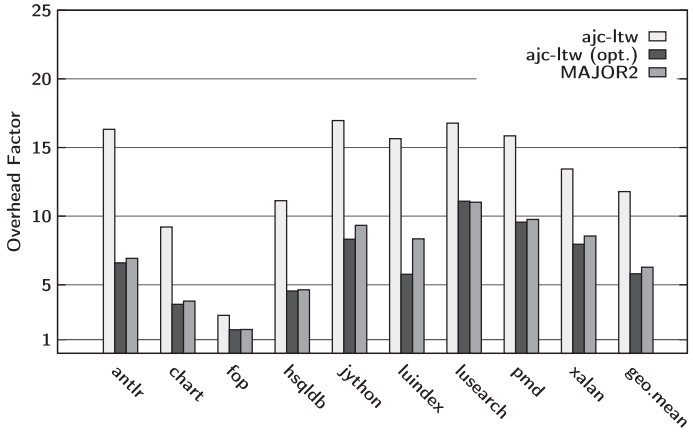

Figure 9. Steady-state overhead for scenario 1.

On average, the start-up overhead (Table III and Figure 10) is lower than the steady-state overhead, as the baseline for comparison consumes much more execution time because of class loading. In the ajc-ltw and ajc-ltw (optimized) settings, the start-up overhead exceeds the steady-state overhead only in the case of fop, a benchmark with short execution time. The most notable difference with steady-state performance is that MAJOR2 causes much more overhead than the optimized

DOI: 10.1002/spe

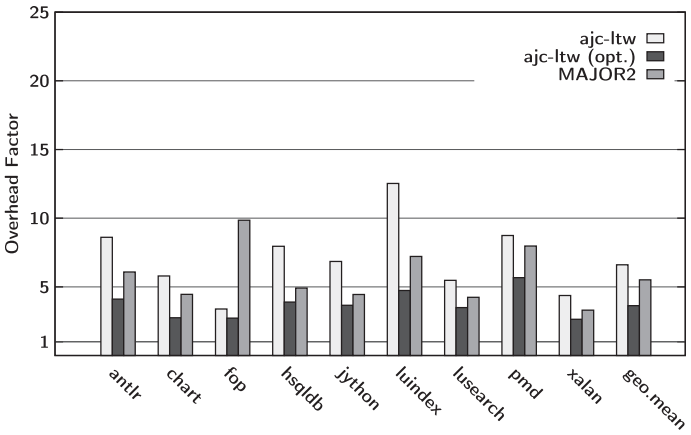

Figure 10. Start-up overhead for scenario 1.

Table IV. Steady-state overhead of dynamic analysis aspects woven with MAJOR2 with complete bytecode coverage.

	Orig.	Scenario 1		Scenario 2	
	[ms]	[ms]	Ovh.	[ms]	Ovh.
antlr	796	10,625	13.35	9644	12.12
bloat	2778	66,935	24.09	53,176	19.14
chart	2863	27,884	9.74	23,548	8.22
eclipse	16,058	85,156	5.30	91,560	5.70
fop	1106	3946	3.57	3507	3.17
hsqldb	2535	16,309	6.43	15,158	5.98
jython	2393	38,194	15.96	31,192	13.03
luindex	3422	40,065	11.71	34,789	10.17
lusearch	1333	17,751	13.32	16,631	12.48
pmd	2396	41,671	17.39	26,094	10.89
xalan	1163	21,713	18.67	20,185	17.36
geo.mean			11.08		9.61

AspectJ version. The straightforward explanation is that the code transformation performed at load time is more expensive with MAJOR2: we call the AspectJ weaver once for each execution level, and then, the code versions are put together in an additional step.

In summary, these results are particularly encouraging for execution levels, which provide much more stable semantics for aspect composition [12, 15]. Our evaluation shows that their efficient integration in a practical aspect language is possible.

8.4.3. Complete bytecode coverage. Our second evaluation measures the overhead introduced by the two profiling aspects woven with MAJOR2 with complete bytecode coverage in both scenarios. That is, the complete Java class library is also woven, as we want to evaluate the overhead of MAJOR2 in concrete scenarios where its novel features are used. A comparison with AspectJ is not possible, because AspectJ is unable to weave the aspects in the Java class library, and is incapable of handling scenario 2. For each benchmark, we report the first run (start-up performance) and we take the median of 15 runs within the same JVM process (steady-state performance). Here, the geometric mean is computed for the whole benchmark suite, including bloat and eclipse, as MAJOR2 is able to handle both correctly.

Tables IV and V show the measured execution times and overhead factors respectively for steady-state and start-up performance. Figures 11 and 12 visualize the overhead factors reported in the tables.

DOI: 10.1002/spe

Table IV presents the results of our measurements for steady-state performance. In the first scenario, the average overhead factor is 11.08, while in the second scenario it is 9.61. The overhead with complete bytecode coverage is almost twice the overhead when weaving only application classes, because Java applications execute large portions of code in methods in the Java class library (as was

Table V. Start-up overhead of dynamic analysis aspects woven with MAJOR2 with complete bytecode coverage.

	Orig.	Scenario 1		Scenario 2	
	[ms]	[ms]	Ovh.	[ms]	Ovh.
antlr	1839	17,261	9.39	17,468	9.50
bloat	4593	77,118	16.79	64,457	14.03
chart	5277	49,937	9.46	48,272	9.15
eclipse	23,673	127,284	5.38	142,605	6.02
fop	1952	23,847	12.22	27,319	14.00
hsqldb	3683	24,088	6.54	23,774	6.46
jython	6886	48,430	7.03	41,101	5.97
luindex	4448	46,103	10.36	39,685	8.92
lusearch	4374	22,415	5.12	22,402	5.12
pmd	4783	58,008	12.13	45,659	9.55
xalan	4475	26,767	5.98	25,987	5.81
geo.mean			8.53		8.11

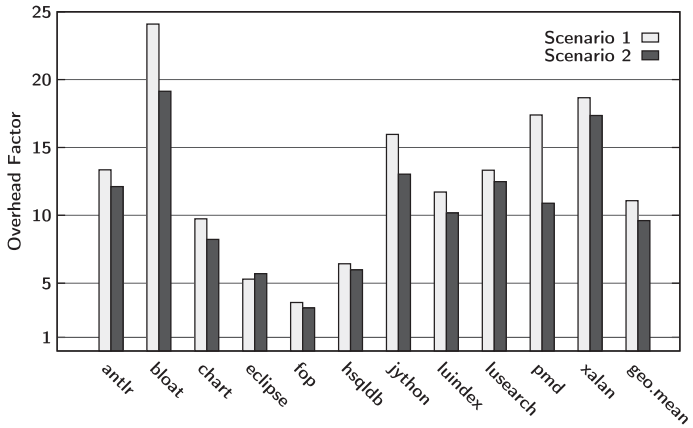

Figure 11. Steady-state overhead for scenarios 1 and 2, MAJOR2 with complete bytecode coverage.

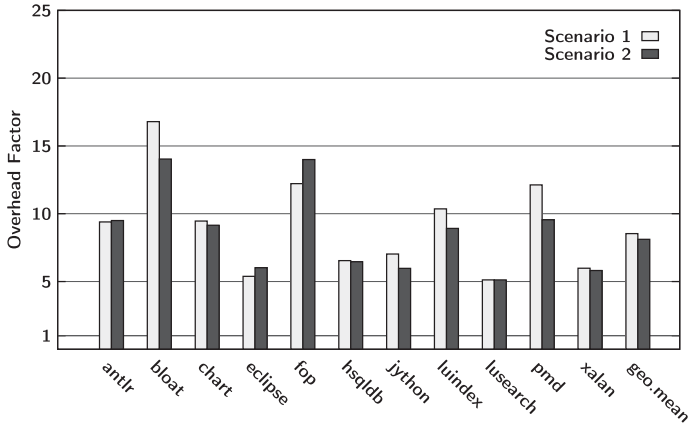

Figure 12. Start-up overhead for scenarios 1 and 2, MAJOR2 with complete bytecode coverage.

DOI: 10.1002/spe

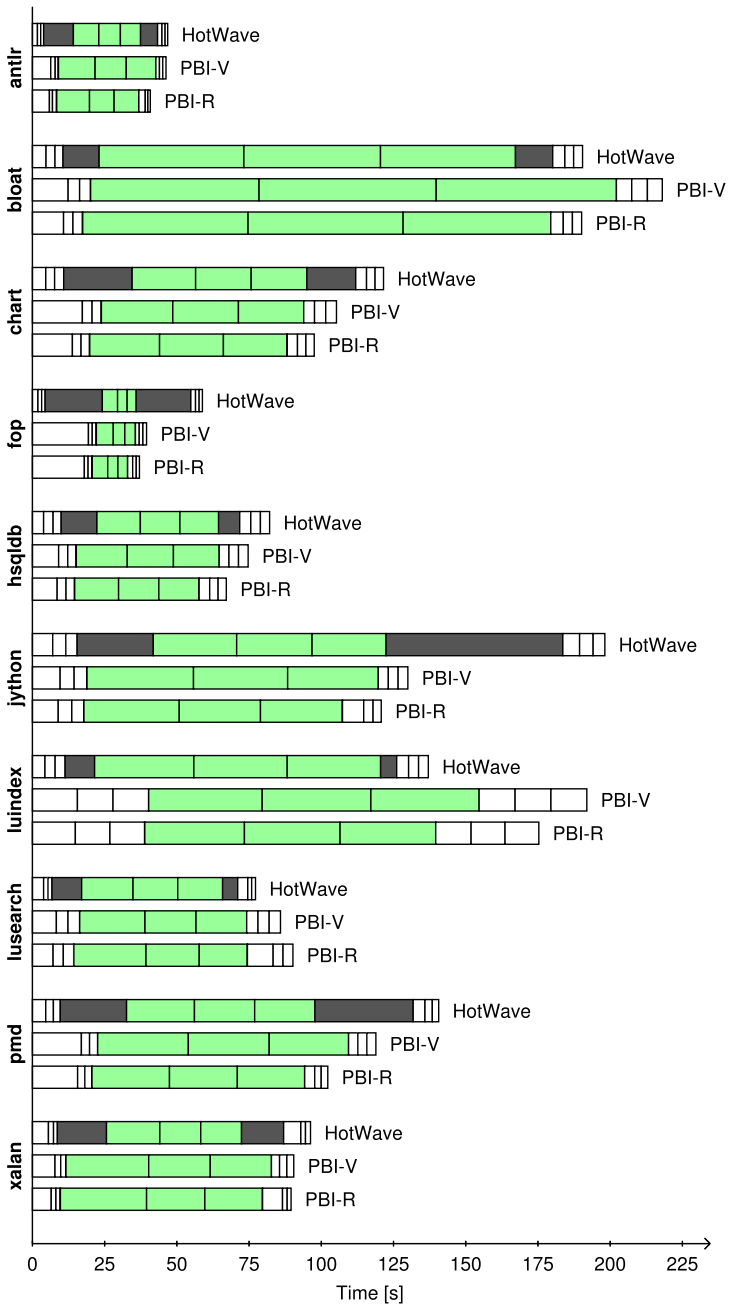

Figure 13. Adaptive dynamic analysis, activating the `ProfCalls` aspect after three benchmark runs and deactivating it after six runs. The dark gray areas illustrate latencies due to runtime weaving and class redefinition with HotWave. PBI, polymorphic bytecode instrumentation.

already pointed out when discussing Table I in Section 8.3). While an overhead factor of 11 is high, it must be considered that the applied instrumentations are computationally expensive. `ProfAllocs` intercepts each object allocation, and `ProfCalls` intercepts each method call. Upon all these intercepted join points, a thread-safe data structure is updated.

Table V illustrates the start-up performance. In the first scenario, the average overhead factor is 8.53, while in the second scenario it is 8.11. As previously discussed in Section 8.4.2, the start-up overhead is lower than the steady-state overhead, because the baseline for comparison executes much longer because of class loading.

DOI: 10.1002/spe

This evaluation confirms that MAJOR2 allows us to create dynamic analysis tools with AOP that have practical value, thanks to complete bytecode coverage. Depending on the concrete analysis, the overhead introduced by complete bytecode coverage can be significant, as many Java applications spend a big part of their execution in methods of the Java class library.

8.5. Adaptive analysis

Finally, we evaluate PBI for adaptive dynamic analysis and assess the cost of PBI-based dispatch compared with class redefinition. Concretely, we compare CodeMerger in its default modus with HotWave [19], a dynamic AOP framework that is based on runtime weaving and class redefinition.[‡‡] We use the `ProfCalls` aspect introduced in Section 8.4.1 as dynamic analysis. We exclude results for the eclipse benchmark, because HotWave excludes many benchmark classes from weaving, similar to ajc-ltw. That is, execution time for eclipse with HotWave would be too short and therefore misleading.

We execute nine runs of the benchmarks within a single JVM process. The first three runs execute original code; then, we activate the dynamic analysis for all classes for three runs, and finally, we execute again original code for the last three runs. For CodeMerger, we present two settings: PBI-V keeps the global state in a volatile field (which is read by the `computeCV()` function), whereas PBI-R uses class redefinition to change the accessor of that field to return a constant, as discussed in Section 7.

Figure 13 shows the execution times as bars with nine segments, one for each run. White segments correspond to runs executed without analysis, and green (or light gray) segments are runs with dynamic analysis. Dark gray areas represent the time spent in runtime weaving and class redefinition. With HotWave runtime weaving and class redefinition may take long time, because all modifiable classes are processed. For instance, with fop, runtime weaving and class redefinition take more than 50% of the overall execution time. jython has the longest redefinition time of 61 s.

In contrast, with CodeMerger, the activation (and deactivation) of the analysis is almost instantaneous, the maximum latency being less than 100 ms in all cases. However, PBI introduces some extra overhead when running original code (without analysis), because of the dispatch switch and code bloat in each method, whereas HotWave introduces no overhead when executing original code. With CodeMerger, the first run is particularly slow because of load-time weaving, which is not needed for HotWave. For some benchmarks, particularly for luindex, the difference in execution time with CodeMerger versus HotWave when executing original code is surprisingly high. The reason is that in this case study we use CodeMerger in its default modus, which can result in high overhead because of increased method size, as explored in Section 8.2.

Note that both HotWave and CodeMerger in the PBI-R setting make use of class redefinition. With Oracle's HotSpot VM, this feature may trigger de-optimization of compiled native code (e.g., undoing method inlining). Consequently, the run that follows class redefinition is often longer than the subsequent runs. Because the HotSpot VM keeps information on hot methods upon class redefinition, the de-optimized code is quickly re-optimized after class redefinition.

Comparing overall execution times for the nine benchmark runs, CodeMerger outperforms HotWave in seven out of 10 benchmarks. For CodeMerger, the PBI-R setting outperforms the PBI-V setting for nine out of 10 benchmarks. Note that these results depend very much on the concrete evaluation settings. On the one hand, if the analysis is frequently activated and deactivated, one can expect that CodeMerger outperforms HotWave because of the dominant overhead of class redefinition. On the other hand, if the analysis is rarely (or never) activated, HotWave may outperform CodeMerger, as HotWave does not incur the overhead of PBI dispatch when the analysis is not woven.

In conclusion, our evaluation confirms that PBI is well suited for building adaptive dynamic analysis tools. As the latency incurred when switching between different code versions is small, adaptive tools built with CodeMerger can quickly react to user choices.

[‡‡] Please note that PBI and HotWave are not functionally equivalent systems. HotWave enables the runtime deployment of instrumentations that may not be available when the base program is started. In contrast, with PBI, all instrumentations must be known in advance.

DOI: 10.1002/spe

9. DISCUSSION

In this section, we first discuss prior and ongoing work by the authors of this article and second compare our approach with related work by others.

9.1. Prior and ongoing work

The PBI generalizes some previously developed techniques. In this section, we give a short overview of our prior research that finally resulted in this proposal.

The FERRARI framework [16] takes any user-defined bytecode instrumentation (which can be implemented with any bytecode manipulation library) and augments it with support for complete bytecode coverage. To this end, FERRARI relies on code duplication within method bodies, similar to the approach presented in Section 3. However, as FERRARI lacks support for merging multiple independent bytecode instrumentations. While the case study presented in Section 5 – the profiler JP2 – could also be implemented with FERRARI, the other two case studies presented in this article may require merging of multiple instrumentations and therefore cannot be handled by FERRARI.

Based on FERRARI, the aspect weaver MAJOR [18] supports most constructs of the AspectJ language and enables aspect weaving with complete bytecode coverage. Thanks to MAJOR, aspect-based dynamic analysis tools, such as profilers [31, 32] or data race detectors [30, 33], are able to analyze all bytecode executed in a JVM. The dynamic AOP framework HotWave [19] relies on the same implementation techniques as FERRARI in order to achieve complete bytecode coverage.

Tanter introduced the notion of *execution levels* as a means to structure aspect-oriented programs so as to prevent infinite regression and unwanted interference between aspects [12]. Attracted by the idea of having execution levels in AspectJ, we developed a first ad hoc implementation [15]. This implementation and the commonalities with the techniques used in FERRARI and MAJOR progressively led us to the formulation of the PBI technique and the implementation of CodeMerger. As discussed in Section 6, PBI enables a clean re-implementation of execution levels for AspectJ. In addition, the PBI-based implementation discussed in this article enables a thorough evaluation with the complete DaCapo benchmark suite, where various compositions of aspects are woven with complete bytecode coverage.

The profiler JP2 used as a case study in this article was first presented in [22, 23]. It has been used for workload characterization at the bytecode level [21]. JP2 is available as an open-source release that includes CodeMerger.

The dynamic program analysis framework DiSL (domain-specific language for instrumentation) [34], available as open-source software (http://disl.ow2.org/), relies on PBI and CodeMerger to ensure analysis with comprehensive bytecode coverage.

9.2. Related work

To the best of our knowledge, there is not much work that is *directly* related to this proposal of PBI. Altering program semantics through bytecode transformations is a widely used technique and has been explored and put in practice in many different flavors in Java, from low-level tools like BIT [35], BCEL [6], and ASM [36], to higher-level frameworks like Javassist [8], Jinline [37], or Soot [38]. Similar toolkits have also been proposed for other languages based on virtual machines that run intermediate bytecodes, like Squeak Smalltalk [9] and .NET. PBI is a general-purpose technique that allows to combine instrumentations possibly written with any of these tools. Thus, it stands at a higher-level than specific instrumentation tools and cannot be directly compared. The most recent version of CodeMerger, our PBI implementation for Java, is implemented using ASM, although other frameworks could be used as well.

On the other hand, there is a huge body of language-level proposals for advanced dispatch, like mixin layers [39], dynamic layer activation [40, 41], aspects [4], and predicate dispatch [42]. Each of these has been realized using particular implementation techniques, specific to the targeted semantics and the implementation trade-offs that their authors were willing to make. Here again, PBI does not stand at the same level as these proposals: PBI is not a language-level mechanism but rather

DOI: 10.1002/spe

an implementation technique to combine various bytecode instrumentations with the possibility to flexibly dispatch among them at runtime. It can be used to implement language-level constructs like mixin layers [13] provided a tool is available to generate the different code versions, or to extend aspect weaving with execution levels (Section 6), again relying on another tool for the specific details of the implementation (in that case, the standard AspectJ weaver).

Shrike [43] is a bytecode instrumentation library that is part of the T.J. Watson Libraries for Analysis [44]. Shrike supports composition of multiple instrumentations by preventing an instrumentation tool from observing the code inserted by the other tools. Similar functionality can be achieved by using execution levels on top of PBI. As described in Section 6, analyses deployed at the same execution level cannot observe each other's join points, exactly as it happens with Shrike. However, execution levels also support the deployment of analyses at different levels (e.g., to profile the execution of another dynamic analysis tool), which is not directly supported by Shrike.

RoadRunner [45] is a framework for composing small and simple analyses for concurrent programs. Each dynamic analysis is essentially a filter over event streams, and filters can be chained. However, it is not possible to combine arbitrary analyses. For example, two analyses that filter (e.g., suppress) events in an incompatible way cannot be combined.

The hyperspace approach [46] allows class fragments to be composed in a coherent whole, using a set of composition operators [47]. The approach is therefore different from PBI because each class version in PBI is a *complete* class, not a fragment of it; dynamic dispatch selects the version that is active at a given point in time, according to any criteria. In that sense, PBI is closer to subject-oriented programming [48] where different *views* of a single class can coexist; implementing subject-oriented programming with PBI is an interesting perspective.

Several researchers discuss the advantages of allowing data passing between different analyses [19, 49, 50]. While PBI does not provide any support for inter-analysis communication (as only a single class version is active at a time), programmers may compose several analyses into a single class version, as it happens with analyses deployed at the same execution level (Section 6). Such composed analyses may communicate with each other through shared data structures.

Regarding instrumentation of shared libraries, the Twin Class Hierarchy [51] replicates the full hierarchy of instrumented classes into a separate package that coexists with the original one. However, in [52], the authors show that class replication limits the applicability of bytecode instrumentation in the presence of native code. Because native code is not modified, calls back into bytecode will target methods in the unmodified class. Thus, this approach does not allow transparent instrumentation of the complete Java class library. In contrast, PBI does not duplicate any class, but relies on code replication within method bodies.

The Arnold–Ryder profiling framework presented in [53] uses code duplication combined with compiler-inserted, counter-based sampling. A second version of the code is introduced, which contains all computationally expensive instrumentation. The original code is minimally instrumented to allow control to transfer in and out of the duplicated code in a fine-grained manner, based on instruction counting. This approach achieves low overhead, as most of the time the slightly instrumented code is executed. Similarly to PBI, this approach merges two different instrumentations. While PBI is a general-purpose, high-level framework that can merge any number of independent bytecode instrumentations, the Arnold–Ryder framework is specialized for sampling profiling and implemented directly within the Jikes RVM. Whereas in PBI, the dispatch logic that determines the code version is customizable and executed only upon method entry, the dispatch logic in the Arnold–Ryder framework is hard coded and enables switching within method bodies depending on the number of executed instructions.

Several approaches have been proposed to perform dynamic program analysis at the level of the virtual machine [54, 55]. While these approaches usually benefit from lower runtime overhead and can access VM-internal information, they require modifications to the VM that may complicate deployment and impair portability. As confirmed in our evaluation presented in Section 8, PBI is compatible with unmodified, production-quality JVMs, which will ease the adoption of PBI by developers of industrial-strength program analysis tools and frameworks.

DOI: 10.1002/spe

10. CONCLUSIONS

Polymorphic bytecode instrumentation is a simple yet very effective technique to combine different instrumentations and select among them dynamically. With PBI, third-party instrumentations of a given class are combined into a single class, where each method uses a user-specified dispatch logic in order to select, at runtime, the code version to execute. Therefore, a PBI framework simply merges code versions and generates the appropriate switch.

We have shown that PBI is an effective technique by illustrating its applicability in a wide range of scenarios: to achieve complete bytecode coverage without disrupting VM bootstrap and avoiding infinite regression, to implement a comprehensive profiler, to implement execution levels for AOP, and to support adaptive dynamic analyses. All case studies have been carried out with CodeMerger, our PBI framework for Java bytecode.

A thorough performance evaluation further shows that PBI can be efficiently implemented. In particular, the pure overhead of the dispatch added by PBI is rather low when just-in-time compilation is enabled. The most efficient modus of PBI is the adaptive one, in which versions are merged into a single body by default, and are implemented as private methods if the merged body would become so large that it would prevent inlining. Our benchmarks of the DaCapo suite confirm that complete bytecode coverage is really crucial for profilers, because a large part of computation happens in the core libraries. Our profiler JP2 is able to produce accurate profiles thanks to PBI. We then demonstrate that execution levels for AOP can be efficiently implemented and are actually more efficient than their brittle equivalent AspectJ idioms, based on control flow checks. Finally, PBI makes it possible to implement adaptive analyses that are more reactive than other systems based on class reloading.

We expect PBI to prove useful in many other cases, such as for implementing advanced dispatch mechanisms and language constructs. A preliminary experience with implementing (a restricted form of) dynamic mixin layers is discussed in [13]; extending it and exploring the implementation of other constructs is one of the main venues for future work.

ACKNOWLEDGEMENTS

The research presented in this article was supported by Oracle (ERO project 1332), by the Swiss National Science Foundation (project CRSII2_136225), and by the European Commission (contract ACP2-GA-2013-605442).

REFERENCES

1. Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM: New York, NY, USA, 2005; 519–538.
2. Steele GL, Jr. A growable language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. ACM: New York, NY, USA, 2006; 505–505.
3. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, Springer-Verlag, 2001; 327–353.
4. Hilsdale E, Hugunin J. Advice weaving in AspectJ. In *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lieberherr K (ed.). ACM Press: Lancaster, UK, March 2004; 26–35.
5. Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.2, Available at: http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html, 2006 [last accessed 1 December 2015].
6. Apache Commons. Apache Commons BCEL, Available at: http://commons.apache.org/proper/commons-bcel/ [last accessed 1 December 2015].
7. Bruneton E, Lenglet R, Coupaye T. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, Grenoble, France, 2002. Available at: http://asm.ow2.org/current/asm-eng.pdf [last accessed 1 December 2015].
8. Chiba S. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, Bertino E (ed.)., Lecture Notes in Computer Science. Springer-Verlag: Sophia Antipolis and Cannes, France, June 2000; 313–336.

DOI: 10.1002/spe

9. Denker M, Ducasse S, Tanter É. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures* 2006; **32**(2-3):125–139.
10. Dmitriev M. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*. ACM Press: Redwood Shores, CA, USA, 2004; 139–150.
11. Avgustinov P, Christensen AS, Hendren LJ, Kuzins S, Lhoták J, Lhoták O, de Moor O, Sereni D, Sittampalam G, Tibble J. Abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*. ACM: Chicago, USA, 2005; 87–98.
12. Tanter É. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*. ACM Press: Rennes and Saint Malo, France, March 2010; 37–48.
13. Moret P, Binder W, Tanter É. Polymorphic bytecode instrumentation. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*. ACM Press: Porto de Galinhas, Brazil, March 2011; 129–140.
14. Gosling J, Joy B, Steele GL, Bracha G. *The Java Language Specification* (Third Edition), The Java Series. Addison-Wesley: Boston, USA, 2005.
15. Tanter É, Moret P, Binder W, Ansaloni D. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*. ACM Press: Eindhoven, The Netherlands, October 2010; 113–122.
16. Binder W, Hulaas J, Moret P. Advanced Java bytecode instrumentation. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. ACM Press: New York, NY, USA, 2007; 135–144.
17. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (second edition). Addison-Wesley: Reading, MA, USA, 1999.
18. Villazón A, Binder W, Moret P. Flexible calling context reification for aspect-oriented programming. In *Proceedings of the 8th international conference on Aspect-Oriented Software Development*, AOSD '09. ACM: Charlottesville, Virginia, USA, March 2009; 63–74.
19. Villazón A, Binder W, Ansaloni D, Moret P. Advanced runtime adaptation for Java. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09. ACM: Denver, Colorado, USA, October 2009; 85–94.
20. Ammons G, Ball T, Larus JR. 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*. ACM: Las Vegas, Nevada, USA, 1997; 85–96.
21. Sewe A, Mezini M, Sarimbekov A, Binder W. DaCapo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In *OOPSLA '11: Proceeding of the 26th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM: New York, NY, USA, 2011; 657–676.
22. Sarimbekov A, Sewe A, Binder W, Moret P, Schoeberl M, Mezini M. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java virtual machine. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ 2011)*, ACM Press, 2011; 11–20.
23. Sarimbekov A, Sewe A, Binder W, Moret P, Mezini M. JP2: call-site aware calling context profiling for the Java virtual machine. *Science of Computer Programming* 2014; **79**(0):146–157.
24. Dutchyn C, Tucker DB, Krishnamurthi S. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming* 2006; **63**(3):207–239.
25. Chiba S, Kiczales G, Lamping J. Avoiding confusion in metacircularity: the meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, vol. 1049, Lecture Notes in Computer Science. Springer-Verlag: Kanazawa, Japan, 1996; 157–172.
26. Masuhara H, Kiczales G, Dutchyn C. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC 2003)*, vol. 2622, Hedin G (ed.)., Lecture Notes in Computer Science. Springer-Verlag: Warsaw, Poland, 2003; 46–60.
27. NetBeans. The NetBeans Profiler Project, Available at: http://profiler.netbeans.org/ [last accessed 1 December 2015].
28. Blackburn SM, Garner R, Hoffman C, Khan AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, ACM Press, New York, NY, USA, October 2006; 169–190.
29. Bodden E, Forster F, Steimann F. Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, (GI-Edition), Lecture Notes in Informatics, Erfurt, Germany, 2006; 49–54.
30. Bodden E, Havelund K. Racer: effective race detection using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM: Seattle, WA, USA, July 2008; 155–165.
31. Pearce DJ, Webster M, Berry R, Kelly PHJ. Profiling with AspectJ. *Software: Practice and Experience* June 2007; **37**(7):747–777.
32. Ansaloni D, Binder W, Villazón A, Moret P. Rapid development of extensible profilers for the Java virtual machine with aspect-oriented programming. In *Wosp/sipew 2010: Proceedings of the 1st Joint International Conference on Performance Engineering*. ACM Press: San Jose, CA, USA, January 2010; 57–62.

DOI: 10.1002/spe

33. Ansaloni D, Binder W, Villazón A, Moret P. Parallel dynamic analysis on multicores with aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*. ACM Press: Rennes and Saint Malo, France, March 2010; 1–12.
34. Marek L, Villazón A, Zheng Y, Ansaloni D, Binder W, Qi Z. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings Conference on Aspect-Oriented Software Development*, AOSD '12. ACM: New York, NY, USA, 2012; 239–250.
35. Lee HB, Zorn BG. BIT: a tool for instrumenting java bytecodes. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, 1997; 73–82.
36. OW2 C. ASM – A Java bytecode engineering library, Available at: http://asm.ow2.org/ [last accessed 1 December 2015].
37. Tanter É, Ségura-Devillechaise M, Noyé J, Piquer J. Altering Java semantics via bytecode manipulation. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, vol. 2487, Batory D, Consel C, Taha W (eds)., Lecture Notes in Computer Science. Springer-Verlag: Pittsburgh, PA, USA, October 2002; 283–298.
38. Vallée-Rai R, Gagnon E, Hendren LJ, Lam P, Pominville P, Sundaresan V. Optimizing Java bytecode using the Soot framework: is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, Paphos, Cyprus, 2000; 18–34.
39. Smaragdakis Y, Batory D. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology* 2002; **11**(2):215–255.
40. Costanza P, Hirschfeld R, De Meuter W. Efficient layer activation for switching context-dependent behavior. In *Proceedings of the Joint Modular Languages Conference (JMLC 2006)*, vol. 4228, Lecture Notes in Computer Science. Springer-Verlag: Oxford, England, September 2006; 84–103.
41. Hirschfeld R, Costanza P, Nierstrasz O. Context-oriented programming. *Journal of Object Technology* 2008; **7**(3):125–151.
42. Millstein T. Practical predicate dispatch. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*. ACM Press: Vancouver, British Columbia, Canada, October 2004; 345–364. ACM SIGPLAN Notices, 39(11).
43. IBM. Shrike Bytecode Instrumentation Library, Available at: http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview [last accessed 1 December 2015].
44. IBM. Watson Libraries for Analysis (WALA), Available at: http://wala.sourceforge.net/wiki/index.php/Main_Page [last accessed 1 December 2015].
45. Flanagan C, Freund SN. The RoadRunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th Workshop on Program Analysis for Software Tools and Engineering*, Toronto, Canada, 2010; 1–8.
46. Ossher HL, Tarr PL. Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology*, vol. 648, Akşit M (ed.)., The Kluwer International Series in Engineering and Computer Science. Kluwer: Dordrecht, the Netherlands, 2001; 293–323.
47. Harrison W, Ossher H, Tarr P. General composition of software artifacts. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, vol. 4089, Löwe W, Südholt M (eds)., Lecture Notes in Computer Science. Springer-Verlag: Vienna, Austria, March 2006; 194–210.
48. Harrison WH, Ossher HL. Subject-oriented programming (a critique of pure objects). In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*. ACM Press: Washington, D.C., USA, October 1993; 411–428. ACM SIGPLAN Notices, 28(10).
49. Click C, Cooper KD. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems* 1995; **17**(2):181–196.
50. Binder W, Ansaloni D, Villazón A, Moret P. Flexible and efficient profiling with aspect-oriented programming. *Concurrency and Computation: Practice and Experience* 2011; **23**(15):1749–1773.
51. Factor M, Schuster A, Shagin K. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM: New York, NY, USA, 2004; 288–300.
52. Tilevich E, Smaragdakis Y. Transparent program transformations in the presence of opaque code. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. ACM: New York, NY, USA, 2006; 89–94.
53. Arnold M, Ryder BG. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, USA, 2001; 168–179.
54. Arnold M, Vechev M, Yahav E. Qvm: an efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08. ACM: New York, NY, USA, 2008; 143–162.
55. Arnold M, Vechev M, Yahav E. Qvm: an efficient runtime for detecting defects in deployed systems. *ACM Transactions on Software Engineering and Methodology* 2011; **21**(1):2:1–2:35.

DOI: 10.1002/spe