Polymorphic Bytecode Instrumentation

Philippe Moret
Faculty of Informatics
University of Lugano
Switzerland
philippe.moret@usi.ch

Walter Binder
Faculty of Informatics
University of Lugano
Switzerland
walter.binder@usi.ch

Éric Tanter*
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
etanter@dcc.uchile.cl

ABSTRACT

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 paper we introduce *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 provide an implementation of execution levels for AspectJ, which avoid infinite regression and unwanted interference between aspects. Second, we present a framework for adaptive dynamic analysis, where the analysis to be performed can be changed at runtime by the user. Third, we describe how PBI can be used to support a form of dynamic mixin layers. We provide thorough performance evaluations with dynamic analysis aspects applied to standard benchmarks. We show 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. We also demonstrate 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.

Categories and Subject Descriptors: D.3.3 Programming Languages: Language Constructs – *Frameworks*

General Terms: Algorithms, Languages, Measurement

Keywords: Bytecode instrumentation, modularity constructs, dynamic program analysis, aspect-oriented programming, mixin layers, 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 [9] or Fortress [32], 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 [24], are implemented using bytecode instrumentation [22]. Because bytecode instrumentation has become 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 (JVMTI)[1] and with the API in the package *java.lang.instrument*. In addition, there are numerous instrumentation libraries for Java bytecode, such as BCEL[2], ASM[3], or Javassist [10], as well as for other languages [14].

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 [15] is a good example of an adaptive profiler.

In this paper we introduce *polymorphic bytecode instrumentation* (PBI), a novel technique that allows several, possibly independent bytecode instrumentations to coexist and be selected dynamically. First, different bytecode instrumentations are applied in isolation to a program class. Afterwards, 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

*Partially funded by FONDECYT Project 1110051.

AOSD'11, March 21–25, 2011, Pernambuco, Brazil.

[1] http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html
[2] http://jakarta.apache.org/bcel/
[3] http://asm.ow2.org/

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 [5].

Polymorphic bytecode instrumentation is a general technique that is applicable to any intermediate language. In this paper we focus on CodeMerger, our PBI framework for Java bytecode. Two of the case studies we consider are related to aspect weaving, and one deals with another modularity construct, mixin layers [31]. The original, scientific contributions of this paper are as follows:

1. We introduce Polymorphic Bytecode Instrumentation, 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 describe three consequent case studies of PBI. First, we use PBI to support *execution levels* [33] in AspectJ (Section 4), thereby enabling black-box composition of dynamic analysis aspects in multiple ways. Second, 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 5). Third, we describe how PBI can be used to support another modularity construct, a form of *dynamic* mixin layers (Section 6).

4. After a brief discussion of technical details (Section 7), we thoroughly evaluate our PBI implementation for Java (Section 8), using the two first case studies. Our evaluation 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; we also show 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 and related work. Section 10 concludes.

2. POLYMORPHIC BYTECODE INSTRUMENTATION

Many tools make use of bytecode instrumentation to achieve different goals. Polymorphic bytecode instrumentation (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[4], as well as modify method bodies, but it may not remove any field or method. Bytecode instrumentations are treated as *black boxes*[5], which may be implemented with any instrumentation framework, not necessarily the same. A PBI framework is in charge of integrating these instrumentations, as explained below.

[4]In this paper "method" stands for "method or constructor".

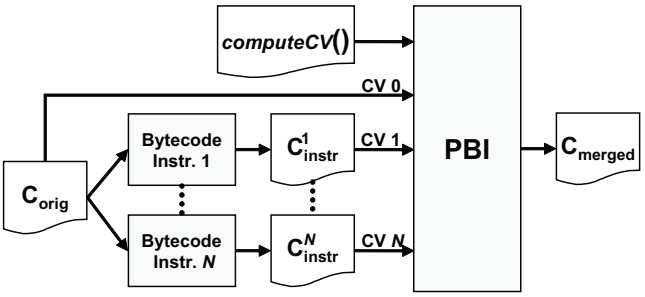

Figure 1: Overview of Polymorphic Bytecode Instrumentation. First, N different bytecode instrumentations are applied to the original class C_{orig} (class version 0), producing the instrumented classes C^i_{instr} (class versions i, $1 \leq i \leq N$). The PBI framework merges the class versions into 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()*.

PBI overview Let us consider $N \geq 1$ bytecode instrumentations that are applied to a base program class C_{orig}. Each black-box instrumentation produces an instrumented class, denoted C^i_{instr} ($1 \leq i \leq N$). 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 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 ($1 \leq i \leq N$).

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.

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.

Dynamic dispatch Support for dynamic dispatch between code versions is at the heart of polymorphic bytecode instrumentation. It offers the necessary flexibility to use PBI in a wide range of scenarios, such as complete bytecode coverage (Section 3), execution levels for AOP (Section 4), adaptive dynamic analysis tools (Section 5), dynamic mixin layers (Section 6), and so forth.

The case studies developed here highlight two different kinds of dispatch logic, based either on (global) state or on control flow.

[5]As we will explain later, 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.

```
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();
```

Figure 2: Code pattern generated by CodeMerger when merging $N+1$ code versions.

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[6]. 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 first two 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. The third case study, which explores support for dynamic mixin layers, illustrates a good scenario for class-specific dispatch.

PBI for Java: CodeMerger Our implementation of PBI for Java bytecode is called CodeMerger. CodeMerger uses BCEL and is implemented in less than 1000 lines of Java code. Each input class version is passed 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 4. 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.

Figure 2 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.

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, 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 before merging, so as to avoid name clashes.

[6]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 [19]).

```
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
      ...
}  }
```

Figure 3: 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.

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 7.

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 [6]. The general technique of polymorphic bytecode instrumentation 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 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.

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 [26]), 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* (Figure 3) maintains that global state in a static volatile flag. The flag is toggled (by an invocation of *signalEndOfBootstrap()*) before the base program

```
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;
}  }
```

Figure 4: Class *ControlFlow* provides access to boolean, thread-local control flow information.

main class is initialized. 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.[7]

While in prior work [6] 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 get rid of that overhead. If the JVM supports class redefinition, 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.

Preventing infinite regression If code inserted by an instrumentation invokes some instrumented methods in the Java class library, infinite regression can happen. 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* (Figure 4) 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, inserted code that may invoke instrumented methods has to use the following code pattern:

```
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 above 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 [38] and HotWave [37] generate the code pattern on code previously woven with the standard AspectJ weaver in a fully automated way.

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;

4. EXECUTION LEVELS FOR AspectJ

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 [16] 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 [11].

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 done by invoking *proceed* in the advice body. The original computation ought to run at the same level at which it originated![8] 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 5. 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 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.

[7]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 VMs and IBM's J9 VMs.

[8]This issue is precisely why using control flow checks in AspectJ in order to discriminate advice computation is actually flawed. See [33] for more details.

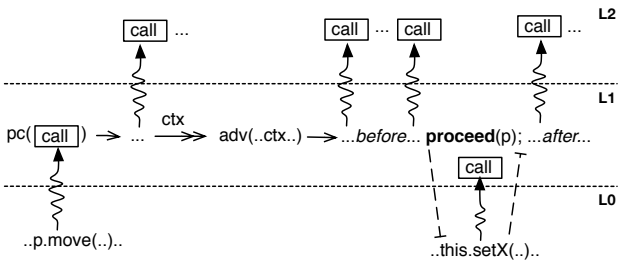

Figure 5: Execution levels in action: pointcut and advice are evaluated at level 1, *proceed* goes back to level 0 (from [33]).

```
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;
}  }
```

Figure 6: Class *ExecutionLevel* provides access to the execution level of the current thread.

Execution levels for AspectJ Execution levels have been formulated and prototyped in aspect languages with dynamic weaving [33]. 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 [27, 22]. The detailed motivation, design, and applications of this extension are presented elsewhere [34]. Our previous implementation of level switching is done 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 [27]: 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 [22]. 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*, *i.e.* a conditional expression that guards the invocation of the advice.

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 (Figure 6). 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 done for the dynamic extent of both advice and pointcut residues, following the given pattern[9]:

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

A similar shift-down occurs for around advice, when the original computation is finally called with *proceed* (see [34] for details).

Our PBI-based aspect weaver, MAJOR2, uses the standard AspectJ weaver as a black box 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:

$$computeCV() \equiv return\ BootstrapState.bootstrapCompleted()\ ?\ ExecutionLevel.currentLevel()\ :\ N;$$

The evaluation in Section 8 uses two different deployment configurations with two aspects in order to assess the performance of the PBI-based implementation.

5. 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, since only data of current interest is gathered.

A good example of an adaptive profiler is JFluid [15], which has been integrated in the NetBeans Profiler[10]. JFluid measures execution time for selected methods and generates a Calling Context Tree (CCT) [1] to help analyze the contributions of direct and indirect callees to the execution time of selected methods. When the user selects different methods for profiling at runtime, JFluid adapts the profiling code accordingly with the aid of class redefinition.

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

[9]Explicit level shifting as proposed in [33] is done following the same pattern.

[10]http://profiler.netbeans.org/

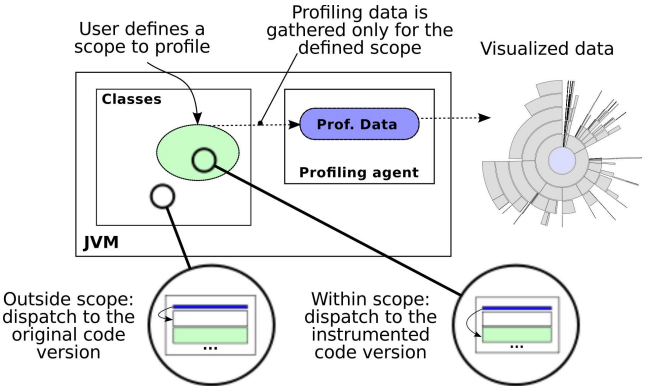

Figure 7: Adaptive dynamic analysis with PBI.

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 [37], which relies on runtime aspect weaving and on class redefinition, weaving an aspect into all modifiable classes at runtime may take up to 60 seconds on a recent machine (see Section 8).

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 [37]. 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 7 illustrates a case of adaptive dynamic analysis with PBI: the user defines, and change dynamically, the *scope* of the profiling; profiling data is 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. 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 below is a template where the meta-variable *selectedCodeVersion* refers to the corresponding volatile field to be read:

$$computeCV() \equiv return\ BootstrapState.bootstrapCompleted()\ \&\&\ !ControlFlow.inCFlow()\ ?\ selectedCodeVersion\ :\ 0;$$

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:

$$computeCV() \equiv return\ selectedCodeVersion;$$

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. Due to the de-optimization and re-optimization caused by class redefinition, changing the volatile variable does introduce some temporary overhead. However, compared to 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).

6. DYNAMIC MIXIN LAYERS

Many mechanisms have been proposed to express *refinements* of classes, such as open classes, mixins, classboxes, inter-type declarations, etc. Mixin layers [31] allow the definition of a set of refinements in a layer, which is the unit of large-scale refinement. Layers gather a number of refinements to classes, such as new methods and fields, as well as overridden methods. The notion of mixin layers has also been recently used in a dynamically-scoped fashion to implement Context-Oriented Programming (COP) languages [23]. COP is a paradigm concerned with providing explicit language support for context-dependent adaptation of programs. A largely adopted mechanism in COP is *dynamic* mixin layers, that is, mixin layers that can be activated dynamically.

ContextL first introduced this mechanism [12]: it allows certain classes to be defined as *layered*, meaning that they can be refined and extended in so-called layers. For instance, the following defines a layered class *Person*, refined in an *Employment* layer, where a field is added and method *display* is refined:

```
class Person {
   String name;
   public String getName() { ... }
   public void display () { ... }
}
layer Employment {
   class Person {
      Person employer;
      public void display () { ... }
   }
}
```

In order to support context adaptation, layers can be dynamically activated for the dynamic extent of the body of *with-active-layer* expressions. For instance:

```
with-active-layer(Employment){ processStaff(); }
```

During the dynamic extent of the *processStaff* call, *Person* objects have an *employer* field and a refined *display* method.

Efficient layer activation in ContextL uses features of Common Lisp that are not present in Java, like multiple inheritance and multiple dispatch [13]. Polymorphic bytecode instrumentation can

```
public class Layers {
   public static int currentLayers() {
      Thread t = Thread.currentThread();
      return t.pbi_layers;
   }
   public static void activate(int layerNr) {
      Thread t = Thread.currentThread();
      t.pbi_layers |= 1 << layerNr;
   }
   public static void deactivate(int layerNr) {
      Thread t = Thread.currentThread();
      t.pbi_layers &= ~(1 << layerNr);
   }
}
```

Figure 8: Class *Layers* provides access to the binary representation of the set of active layers in the current thread.

be used to support efficient dynamic layer activation in a Java-like language, without using inheritance. We focus on the core mechanism—dynamic mixin layer activation—without considering many additional useful features supported by full-blown mixin layers and COP languages, like invoking methods in upper layers, embedding layer definitions within classes, and so on [31, 23]. The purpose of this section is to describe how the PBI technique, and in particular the dispatch mechanism, can also handle this scenario.

Dynamic mixin layers with PBI The general approach consists in obtaining one class version per *combination* of layers from an external layer compiler.[11] More precisely, if k is the number of layers that refine class C, we need 2^k class versions. In merged methods, the dispatch logic, which is now *specific to the considered class*, obtains the currently-active layers and switches to the corresponding code version.

Concretely, all layers are given a number, and for this proof-of-concept implementation, we limit the number of layers to 32, so we can represent the set of currently-active layers with a 32-bit integer. Class *Layers* in Figure 8 gives access to the thread-local set of currently-active layers, stored in the *pbi_layers* fields of threads, similarly to how execution levels are handled (Figure 6). Its *activate()* and *deactivate()* methods actualize the binary representation of the active layers. Activating a layer for a certain dynamic extent is done following the same pattern as for execution level shifting (Section 4).

For each class, we know statically what layers can refine it (this is explicit in layer definitions); therefore, we associate a bitmask that specifies which layers may affect the class. The bit-wise `&` operation between the currently-active layers and the class bitmask gives us the appropriate version number. That is, for a class C, the dispatch logic is:

computeCV() ≡ *return Layers.currentLayers() & Mask_C;*

where *Mask_C* is the bitmask where the bits are set that correspond to the numbers of the layers that refine class C.

For instance, if 10 layers are defined (0 to 9) and class C is refined in layers 0 and 3, the bitmask of C is `1001` (or, `0x9`). With the layer compiler, we obtain the class versions 1 (where layer 0 is woven), 8 (where layer 3 is woven) and 9 (where both layers 0 and 3 are woven). Given these versions, as well as class version 0 (the original class definition) and the class-specific dispatch, the generated switch is as follows:

```
int version = Layers.currentLayers() & 0x9;
switch (version) {
   case 0: goto CV0; // no layer is active
   case 1: goto CV1; // layer 0 is active
   case 8: goto CV8; // layer 3 is active
   case 9: goto CV9; // layers 0 and 3 are active
   default: goto Error;
}
```

Obviously, the point of this case study is not to present a full-fledged dynamic mixin layers mechanism (an interesting perspective that is left for future work). Rather, it shows that the customizable dispatch mechanism of PBI can be based not only on global or thread-local state, but also on class-specific information. Of course, it could also be based on object-specific state, although we have not explored this possibility yet.

In terms of efficiency, the cost of the dispatch used in this study is the same as that of execution levels (see Section 8), plus the cost of the bit-wise `&` operation, which turns out to be negligible.

7. CODEMERGER: DETAILS

In this section we describe some technical details and limitations of CodeMerger, our implementation of PBI for Java. In particular, we explain the overall process of applying PBI with complete bytecode coverage, and discuss issues regarding the initialization of inserted fields.

Build-time and load-time instrumentation With CodeMerger, the Java class library is instrumented at build-time before running an (instrumented) application, whereas all other classes are instrumented at load-time. Load-time instrumentation in pure Java is supported by the *java.lang.instrument* API. CodeMerger does not instrument the Java class library at load-time, because this would require either native code (using the JVMTI) or redefinition (hotswapping) of the classes loaded during JVM bootstrapping. The former approach would compromise portability, while the latter approach would seriously restrict the possible bytecode instrumentations, because in current standard JVMs, class redefinition may only replace method bodies, but must not introduce any new methods or fields.

Initialization of static fields According to the code pattern illustrated in Figure 2, 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 Figure 2 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, *e.g.,* if final fields are not initialized. 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 [22].

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

[11] We use a simple BCEL instrumentation in our proof-of-concept implementation.

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.

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 [26], 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, because custom class-loaders may not be able to find or may refuse to load the extra classes. However, for load-time instrumentation, CodeMerger does not introduce any extra classes (load-time instrumentation does not happen before JVM bootstrap completes).

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. As an alternative, post-instrumentation transformations can be done: for instance, in the case of the AspectJ weaver, 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.

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.

Code bloat The JVM specification [26] imposes several restrictions on class files. 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 issue. This issue can be mitigated by placing code versions in separate private methods when the method size limit is exceeded.

8. EVALUATION

In this section we evaluate PBI focusing on the first two case studies (Section 4 and Section 5).

Settings Our evaluation is done with two bytecode instrumentations for dynamic program analysis implemented as aspects, the object allocation profiler *ProfAllocs* and the method call profiler *ProfCalls* (Figure 9). The allocation profiler collects the number of object allocations for each type, and the method call profiler collects the number of method calls for each method. Both profilers maintain a thread-safe mapping from identifiers to atomic integers (methods *profileAllocation(...)* and *profileCall(...)*, which are not shown in the figure). For *ProfAllocs*, the identifiers are the types represented by *java.lang.Class* instances, and for *ProfCalls*, the method identifiers are represented by *JoinPoint.StaticPart* instances. We are using non-blocking data structures from the *java.util.concurrent* package, concretely *ConcurrentHashMap* and *AtomicLong*. We discuss the scoping pointcuts defined in *ScopeProf* below.

```
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());
}
```

Figure 9: Simplified aspects for object allocation and method call profiling.

We weave the *ProfAllocs* and *ProfCalls* aspects in the DaCapo benchmarks (dacapo-2006-10-MR2), which serve as base programs. For the first case study, we use our new PBI-based reimplementation of MAJOR2 [34] that relies on CodeMerger, which provides support for execution levels and complete bytecode coverage. For the second case study, we directly use CodeMerger. In both case studies, aspects are woven with AspectJ 1.6.5 (MAJOR2 is also based on AspectJ).

Our measurement machine is a quad-core machine (Dell Optiplex 760, 1 quad-core Intel CPU, 3.0 GHz, 4 GB RAM) running Fedora 13 and the Oracle JDK 1.6.0_18 Hotspot Server VM (64 bit version with default settings). This JVM has one of the most advanced just-in-time compiler performing various code optimizations at runtime, which helps reduce the overhead introduced by PBI dispatch logic.

Execution levels 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*.

Our first evaluation compares the performance of code woven with AspectJ's load-time weaver (henceforth called ajc-ltw) versus MAJOR2. In this comparison we use scenario 1, since 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 take the median of 15 runs executed in the same JVM process.

Table 1: Overhead comparison between AspectJ and MAJOR2 in scenario 1. Aspects are woven only into application classes.

	Orig.	ajc-ltw		ajc-ltw (opt.)		MAJOR2	
	[ms]	[ms]	Ovh.	[ms]	Ovh.	[ms]	Ovh.
antlr	798	12416	**15.56**	4926	**6.17**	5000	**6.27**
chart	2867	25906	**9.04**	10132	**3.53**	10650	**3.71**
fop	1129	3032	**2.69**	1799	**1.59**	1871	**1.66**
hsqldb	2614	28720	**10.99**	11570	**4.43**	10983	**4.20**
jython	2390	40183	**16.81**	19435	**8.13**	16380	**6.85**
luindex	3453	53212	**15.41**	21229	**6.15**	27921	**8.09**
lusearch	1370	21359	**15.59**	14368	**10.49**	14139	**10.32**
pmd	2373	37995	**16.01**	22420	**9.45**	14597	**6.15**
xalan	1100	15998	**14.54**	9360	**8.51**	9443	**8.58**
geo.mean			**11.63**		**5.71**		**5.54**

We also compute the geometric mean for all benchmarks except bloat and eclipse, the first because it fails with ajc-ltw (in contrast to MAJOR2) and the latter because ajc-ltw fails to weave a large number of classes due to dependencies. That is, ajc-ltw depends on classes that are also used by the eclipse benchmark. Such problem does not exist with MAJOR2, which makes proper use of class-loader namespaces.

Aspects in Figure 9 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 [7] and is used in many aspect implementations [8]. 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 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.

Table 1 shows the measured execution times and overhead factors. MAJOR2 introduces significantly less overhead than ajc-ltw (factor 5.54 for MAJOR2 versus factor 11.63 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 regression and aspect interferences.

It is surprising that MAJOR2 introduces slightly less overhead than the optimized ajc-ltw case (factor 5.71 on average). This result is due to non-determinism and heuristics used in the optimizing just-in-time compiler. It is possible that some code pattern triggers or prevents specific optimizations; this issue in benchmarking Java software is well known [18]. Whether ajc-ltw (with optimized pointcuts) or MAJOR2 performs better depends a lot of the concrete benchmark. For example, ajc-ltw is considerably faster than MAJOR2 on luindex, whereas MAJOR2 outperforms ajc-ltw on pmd. We validate soundness of MAJOR2 by comparing the profiles produced with ajc-ltw and with MAJOR2, which are almost identical in most cases (relative difference of both the accumulated object allocations and the method calls is below 0.5% for all benchmarks, and even orders of magnitude smaller for most benchmarks). Some differences are always possible because of non-determinism in the application or in the Java runtime (*e.g.,* identity hash-codes are random values on many JVMs and may change in each benchmark run, thread scheduling is non-deterministic, etc.). In all cases where the relative difference exceeds 0.001%, we also compared the profiles for consecutive runs with the same tool (*i.e.* with ajc-ltw respectively with MAJOR2), and obtained relative differences in the same order of magnitude.

Table 2: Overhead of dynamic analysis aspects woven with MAJOR2 with complete bytecode coverage.

	Orig.	Scenario 1		Scenario 2	
	[ms]	[ms]	Ovh.	[ms]	Ovh.
antlr	798	10016	**12.55**	10123	**12.69**
bloat	2694	57537	**21.36**	59821	**22.21**
chart	2867	26072	**9.09**	25405	**8.86**
eclipse	14985	75137	**5.01**	93432	**6.24**
fop	1129	3809	**3.37**	3678	**3.26**
hsqldb	2614	15978	**6.11**	15484	**5.92**
jython	2390	31912	**13.35**	31851	**13.33**
luindex	3453	37655	**10.91**	37276	**10.80**
lusearch	1370	17258	**12.60**	17710	**12.93**
pmd	2373	29429	**12.40**	27352	**11.53**
xalan	1100	21110	**19.19**	20703	**18.82**
geo.mean			**10.09**		**10.18**

In summary, these results are particularly encouraging for execution levels, which provide much more stable semantics for aspect composition [33, 34], as this shows that their efficient integration in a practical aspect language is possible.

Our second evaluation measures the overhead introduced by the two profiling aspects woven with MAJOR2 with complete method coverage in both scenarios. That is, the complete Java class library is also woven. 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 take the median of 15 runs within the same JVM process. Here, the geometric mean is computed for the whole benchmark suite, including bloat and eclipse, since MAJOR2 is able to handle both correctly.

Table 2 presents the results of our measurements. In both scenarios, the average overhead is about factor 10. The overhead with complete bytecode coverage is almost twice the overhead when weaving only application classes, because Java applications spend much execution time in methods in the Java class library. While overhead factor 10 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.

This evaluation confirms that MAJOR2 allows us to create dynamic analysis tools with AOP that have practical value, because of complete bytecode coverage. The overhead introduced by complete bytecode coverage is significant but not prohibitive, and depends on the concrete analysis.

Adaptive analysis We now evaluate PBI for adaptive dynamic analysis and assess the cost of PBI-based dispatch compared to class redefinition. Concretely, we compare CodeMerger with HotWave [37], a dynamic AOP framework which is based on runtime weaving and class redefinition. With CodeMerger, we achieve complete bytecode coverage, whereas with HotWave a few non-modifiable system classes cannot be redefined. We use the *ProfCalls* aspect 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

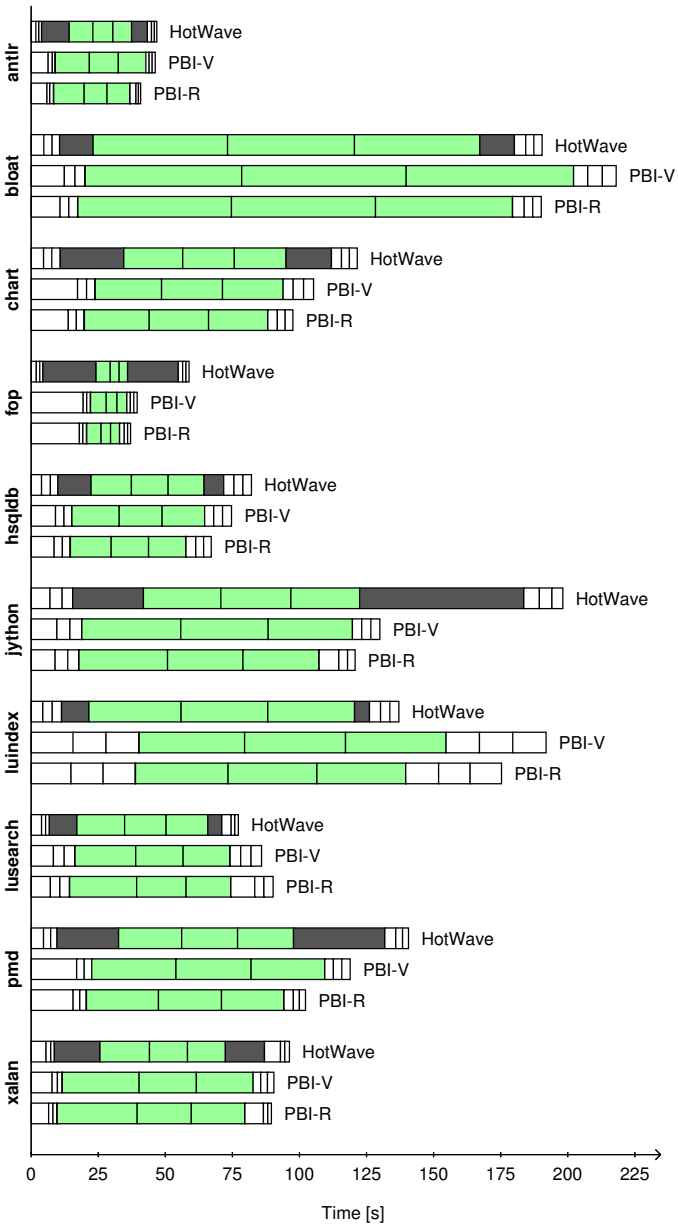

Figure 10: Adaptive dynamic analysis, activating the *ProfCalls* aspect after 3 benchmark runs, and deactivating it after 6 runs. The dark gray areas illustrate latencies due to runtime weaving and class redefinition with HotWave.

for eclipse with HotWave would be too short and therefore misleading.

We execute 9 runs of the benchmarks within a single JVM process. The first 3 runs execute original code, then we activate the dynamic analysis for all classes for 3 runs, and finally we execute again original code for the last 3 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 5.

Figure 10 shows the execution times as bars with 9 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 61s.

In contrast, with CodeMerger the activation (and deactivation) of the analysis is almost instantaneous, the maximum latency being less than 100ms 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. We are currently investigating why the just-in-time compiler does not better optimize the code produced by CodeMerger in these cases.

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. Since 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 9 benchmark runs, CodeMerger outperforms HotWave in 7 out of 10 benchmarks. For CodeMerger, the PBI-R setting outperforms the PBI-V setting for 9 out of 10 benchmarks. 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.

9. DISCUSSION

Prior work 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 [6] 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, none of the case studies presented in this paper can be implemented with FERRARI.

Based on FERRARI, the aspect weaver MAJOR [38] 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 [30, 3] or data race detectors [8, 2], are able to analyze all bytecode executed in a JVM. The dynamic AOP framework HotWave [37] 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 [33]. Attracted by the idea of having execution levels in AspectJ, we developed a first ad-hoc implementation [34]. 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 4, PBI enables a clean re-implementation of execution levels for AspectJ. In addition, the PBI-based implementation discussed in this paper enables a thorough evaluation with the complete DaCapo benchmark suite, where various compositions of aspects are woven with complete bytecode coverage.

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 [25], BCEL, and ASM, to higher-level frameworks like Javassist [10] or Soot [36]. Similar toolkits have also been proposed for other languages based on virtual machines that run intermediate bytecodes, like Squeak Smalltalk [14] 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. CodeMerger, our PBI implementation for Java, is implemented using BCEL, 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 [31], dynamic layer activation [13, 23], aspects [22], predicate dispatch [28], and so on. Each of these has been realized using particular implementation techniques, specific to the targeted semantics and the implementation tradeoffs 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 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 (Section 6) provided a tool is available to generate the different code versions, or to extend aspect weaving with execution levels (Section 4), again relying on another tool for the specific details of the implementation (in that case, the standard AspectJ weaver).

The Hyperspace approach [29] allows class fragments to be composed in a coherent whole, using a set of composition operators [20]. 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 [21] where different *views* of a single class can coexist; actually, implementing subject-oriented programming with PBI is an interesting perspective.

Regarding instrumentation of shared libraries, the Twin Class Hierarchy [17] replicates the full hierarchy of instrumented classes into a separate package that coexists with the original one. However, in [35] 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 [4] 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.

10. CONCLUSION

Polymorphic bytecode instrumentation (PBI) is a simple yet effective technique to combine different instrumentations and select among them dynamically. It is *simple* because it relies on a user-specified, customizable dispatch logic that is in charge of selecting code versions produced by the different instrumentations; 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 full bytecode coverage without disrupting VM bootstrap and avoiding infinite regression, to implement execution levels for AOP, to support adaptive dynamic analyses, and to allow dynamic mixin layer activation. Thorough performance evaluation further shows that PBI can be efficiently implemented. All case studies have been carried out with CodeMerger, our PBI framework for Java bytecode. We expect PBI to prove useful in many other cases, such as for implementing other advanced language constructs; this is one of the main venues for future work.

Acknowledgments The work presented in this paper has been supported by the Swiss National Science Foundation.

11. REFERENCES

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

[2] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 1–12, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[3] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Rapid Development of Extensible Profilers for the Java Virtual Machine with Aspect-Oriented Programming. In *WOSP/SIPEW 2010: Proceedings of the First Joint International Conference on Performance Engineering*, pages 57–62. ACM Press, Jan. 2010.

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

[5] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[6] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM Press.

[7] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition, 2006.

[8] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, July 20-24 2008*, pages 155–165, New York, NY, USA, 07 2008. ACM.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 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*, pages 519–538, New York, NY, USA, 2005. ACM.

[10] S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

[11] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 1996.

[12] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *ACM Dynamic Language Symposium (DLS 2005)*, San Diego, CA, USA, Oct. 2005.

[13] P. Costanza, R. Hirschfeld, and W. De Meuter. Efficient layer activation for switching context-dependent behavior. In *Proceedings of the Joint Modular Languages Conference (JMLC 2006)*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103, Oxford, England, Sept. 2006. Springer-Verlag.

[14] M. Denker, S. Ducasse, and É. Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

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

[16] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

[17] M. Factor, A. Schuster, and K. Shagin. 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*, pages 288–300, New York, NY, USA, 2004. ACM.

[18] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. of OOPSLA'07*, pages 57–76. ACM, 2007.

[19] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.

[20] W. Harrison, H. Ossher, and P. Tarr. General composition of software artifacts. In W. Löwe and M. Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, Vienna, Austria, Mar. 2006. Springer-Verlag.

[21] W. H. Harrison and H. L. Ossher. 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)*, pages 411–428, Washington, D.C., USA, Oct. 1993. ACM Press. ACM SIGPLAN Notices, 28(10).

[22] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 26–35, Lancaster, UK, Mar. 2004. ACM Press.

[23] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March-April 2008.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.

[25] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting java bytecodes. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[27] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC 2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[28] T. Millstein. Practical predicate dispatch. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 345–364, Vancouver, British Columbia, Canada, Oct. 2004. ACM Press. ACM SIGPLAN Notices, 39(11).

[29] H. L. Ossher and P. L. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In M. Akşit, editor, *Software Architectures and Component Technology*, volume 648 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer, 2001.

[30] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.

[31] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, Apr. 2002.

[32] G. L. Steele, Jr. A growable language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 505–505, New York, NY, USA, 2006. ACM.

[33] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 37–48, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[34] É. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 113–122, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

[35] E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 89–94, New York, NY, USA, 2006. ACM.

[36] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[37] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Advanced Runtime Adaptation for Java. In *GPCE '09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, pages 85–94. ACM, Oct. 2009.

[38] A. Villazón, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-Oriented Programming. In *AOSD '09: Proceedings of the 8th International Conference on Aspect-oriented Software Development*, pages 63–74, Charlottesville, Virginia, USA, Mar. 2009. ACM.