Flexible and Efficient Measurement of Dynamic Bytecode Metrics

Walter Binder Jarle Hulaas

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract

Code instrumentation is finding more and more practical applications, but the required program transformations are often difficult to implement, due to the lack of dedicated, high-level tools. In this paper we present a novel instrumentation framework that supports the partial evaluation of compiled Java code transformation templates, with the goal of efficiently measuring chosen dynamic bytecode and control flow metrics. This framework, as well as the instrumentation code it generates, is implemented in pure Java and hence completely platform-independent. We show the benefits of our approach in several application areas, such as platform-independent resource management and profiling of software components.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.2.8 [*Software Engineering*]: Metrics—Performance measures

General Terms Algorithms, Languages, Measurement

Keywords Java, JVM, bytecode instrumentation, program transformations, component-based software engineering, partial evaluation, dynamic metrics, resource management, profiling, aspect-oriented programming

1. Introduction

As Java and the Java Virtual Machine (JVM) are a preferred programming language and deployment platform for many application and middleware developers, there is a need for efficient profiling tools that help analyzing runtime characteristics of Java-based systems. The Java Virtual Machine Profiling Interface (JVMPI) [28] and its successor, the JVM Tool Interface (JVMTI) [29], provide a set of hooks, which signal events inside the JVM (e.g., thread start, method invocation, object allocation, etc.) to profiling tools. However, the interception of such events often causes excessive overhead – up to factor 4 000 during our experiments with profiling based on the JVMPI or JVMTI [5] – or are more generally not appropriate for production-time deployment.

With the aid of bytecode instrumentation, programs can receive minimalistic extensions that collect targeted execution statistics while causing only moderate overhead. Following such program transformation techniques, JVM bytecode instruction (from now on, we will simply write 'bytecode') sequences are inserted into programs at well-chosen locations in order to measure dynamic metrics [15]. In particular, bytecode metrics, such as e.g. the total number of executed bytecodes, can be obtained in this way, while also preserving platform independence [15]. In [5] further advantages of bytecode metrics for profiling are highlighted, including reduced measurement perturbation, reproducibility of measurements, and independence of any platform-specific features; the latter helps implementing instrumentation tools in pure Java. Based on the JVMPI, the JVMTI has partly followed this evolution by incorporating some support for bytecode instrumentation.

Currently, these program instrumentation schemes imply resorting to low-level bytecode engineering frameworks, such as e.g. BCEL [14]. While these do offer bytecode instrumentation without limits, they demand a deep understanding of the JVM, and it takes a significant programming effort to implement the desired program transformations. Tools for aspect-oriented programming in Java, such as AspectJ [19], do usually not match up to the intended bytecode instrumentation, because they identify only higher-level pointcuts, such as method invocations, whereas the collection of bytecode metrics we address here relies on processing at the level of basic blocks.

One important use case for platform-independent profiling is in the area of *service-oriented architectures* (SOA). SOA aim at the construction of applications by integrating advanced service components [25], such as service repositories, matchmakers, service composition and orchestration engines, reputation mechanisms, etc. These components are typically deployed in heterogeneous environments, which means that the actual target platforms are often not known at development time. Therefore, it becomes necessary to have platform-independent profiling support that allows the developer to detect algorithmic inefficiencies during early development phases. Moreover, as components may involve complex algorithms, such as planners for automated service composition, the measurement overhead has to be low in order for such profiling tools to be applicable. Existing profilers are not appropriate for such a setting because of their high overhead, their exclusive focus on highly platform-specific metrics (e.g., CPU time on a particular target system), and frequent suffering from strong measurement perturbation. In contrast, the approach presented here allows to build efficient profiling tools that compute platform-independent dynamic metrics with small implementation effort.

The contribution of this paper is a novel approach to bytecode instrumentation that is based on *code templates* being inserted at well-defined program locations, which themselves are identified by a customizable basic block analysis. Our technique aims at reconciling ease of use and high productivity with the flexibility needed to implement instrumentation schemes for collecting bytecode metrics. We demonstrate our approach with a generic meta-tool called the *Bytecode Metrics Warrior (BMW)*, which allows the developer

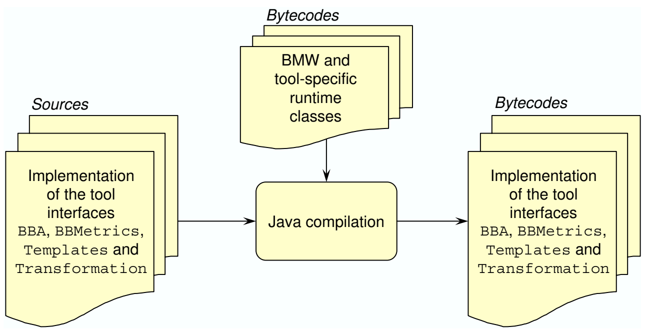
Bytecodes
BMW and tool-specific runtime classes
Sources
Implementation of the tool interfaces BBA, BBMetrics, Templates and Transformation
Java compilation
Bytecodes
Implementation of the tool interfaces BBA, BBMetrics, Templates and Transformation

Figure 1. Step 1: Preparation of transformation (BMW offers 2 `BBA` implementations)

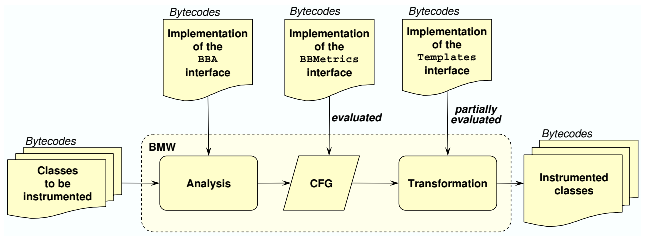
Bytecodes
Implementation of the BBA interface
Bytecodes
Implementation of the BBMetrics interface
Bytecodes
Implementation of the Templates interface
evaluated
partially evaluated
Bytecodes
Classes to be instrumented
BMW
Analysis
CFG
Transformation
Bytecodes
Instrumented classes

Figure 2. Step 2: Program transformation (BMW execution)

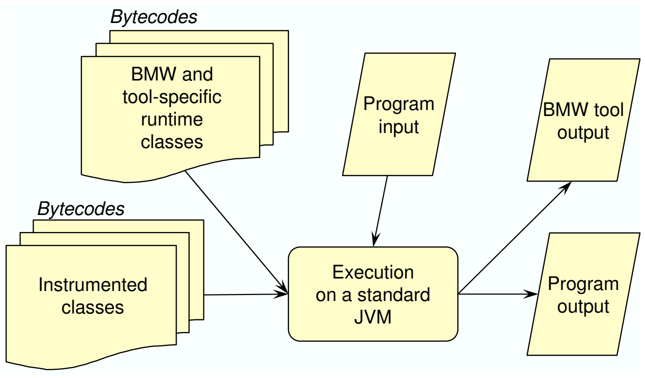

Figure 3. Step 3: Execution of instrumented program

to implement many non-trivial bytecode instrumentation schemes – amongst others for profiling [5, 6] and for resource management [7, 8, 18] – with only a few lines of code. In short, BMW represents a synthesis of our previous work, which it enhances by separating generic bytecode and control flow processing from mission-specific instrumentation schemes, and by proposing high-level programming facilities such as partial evaluation.

This paper is structured as follows: In Section 2 we explain the principles underlying our approach. Section 3 presents our API for defining program transformation schemes and details how BMW instruments programs according to a given scheme. In Section 4 we illustrate our approach with several examples. Section 5 compares our approach with related work. Section 6 discusses the strengths, limitations, and possible extensions of our approach. Finally, Section 7 concludes this paper.

2. BMW Overview

BMW aims at reducing the effort of writing instrumentation tools to compute dynamic bytecode metrics. In a typical use case, such metrics are computed by dynamically aggregating the statically pre-computed metrics for each basic block that is executed. In the following we summarize the design goals underlying BMW:

- Easy to learn, no new programming language. Program transformations are defined in the form of templates, which are written in Java and compiled to bytecode. The bytecode of compiled templates is merged into the program to be transformed at well-defined locations. The merging algorithm relies on the partial evaluation of templates; it has been designed to avoid complex, time-consuming analysis of template code in order to enable efficient instrumentation tools.
- Integration with an existing, well established bytecode instrumentation library in order to avoid implementing low-level operations from scratch. BMW is based on the bytecode engineering library BCEL [14], which has been successfully used in many projects.
- Compact but flexible API to easily define different program instrumentation schemes, focusing on the computation of dynamic bytecode metrics.
- Customizable specification of dynamic bytecode metrics and of what constitutes a basic block. Support for collecting calling context-sensitive dynamic metrics. Instrumentation takes place in the beginning of each basic block. Support for additional instrumentation of method entries and of exception handlers.
- Thread-local data structures, which may e.g. store calling context-sensitive dynamic metrics, are passed as extra arguments in method[1] invocations. As shown in previous work [5, 6, 7, 8, 18], this technique is essential to achieve low overhead.
- Compatibility with native code. As native code is not aware of extra method arguments, wrapper methods with unmodified signature have to establish compatibility between transformed Java code and unchanged native code.

Figures 1, 2, and 3 illustrate the process of creating a BMW-based bytecode instrumentation tool. First (see Figure 1), the user provides implementations of several interfaces, which will be described below, and compiles them with a standard Java compiler. Next (see Figure 2), BMW is executed in order to instrument a given program: A basic block analysis (`BBA` implementation) is used to generate a control flow graph `CFG`. An implementation of the `BBMetrics` interface statically computes bytecode metrics for each basic block in the control flow graph. The bytecode metrics for individual basic blocks constitute the basis for the computation of dynamic bytecode metrics for the execution of a program. The implementation of the `Templates` interface provides instrumentation templates to compute dynamic bytecode metrics at execution time. These templates are partially evaluated and inserted into the program to be instrumented. Finally (see Figure 3), the instrumented program is executed in order to compute the desired bytecode metrics, in addition to the normal program output.

The user of BMW has to provide an implementation of the `Transformation` interface (see Figure 4), which defines all aspects of the desired program transformation. A transformation consists of 3 parts:

- `BBA`: An implementation of the `BBA` interface provides a custom basic block analysis (see Figure 5). It receives the bytecode of a method and computes a control flow graph, in which each node represents a basic block. BMW comes with 2 different `BBA` implementations, which may be directly reused or extended. In addition, the user may implement the desired basic block analysis from scratch.

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

```
// bundles all aspects of a transformation
public interface Transformation {
  BBA getBBA(); // basic block analysis algorithm
  BBMetrics getBBMetrics(); // BBMetrics implementation
  Templates getTemplates(); // program transformation templates
}
```

Figure 4. `Transformation` interface

```
public interface BBA { // basic block analysis
  // Computes a control flow graph for a given method.
  // MethodGen is provided by BCEL.
  CFG computeCFG(MethodGen m);
}

public interface CFG { // control flow graph for a method
  BB getFirstBB(); // first basic block in a method
  BB[] getHandlerBBs(); // first b.b. in each exception handler
  BB[] getAllBBs(); // all basic blocks in a method
  ...
}

public interface BB { // basic block
  // Returns a handle to the first instruction in the basic block.
  // InstructionHandle is provided by BCEL.
  InstructionHandle getFirstInstr();

  int getSize(); // number of bytecodes in the basic block
  ...
}
```

Figure 5. Interfaces for basic block analysis, control flow graph, and basic block

```
public interface BBMetrics {
  // computes bytecode metrics for a given basic block
  int[] getMetrics(BB bb);
}
```

Figure 6. `BBMetrics` interface

```
// program transformation templates
public interface Templates {
  // returns type descriptors of extra arguments
  String[] getExtraArgTypes();

  // inserted in beginning of each method
  void onMethodEntry(MID m);

  // inserted in beginning of each exception handler
  void onHandlerEntry(MID m);

  // inserted in beginning of each basic block
  void onBBEntry(MID m, int[] bbMetrics);

  // inserted in beginning of wrapper methods
  void onWrapper(MID m);
}

// represents method identifiers
public interface MID {}
```

Figure 7. `Templates` and `MID` interfaces

- `BBMetrics`: An implementation of the `BBMetrics` interface computes bytecode metrics for each basic block (see Figure 6).
- `Templates`: An implementation of the `Templates` interface provides the program transformation templates (see Figure 7).

3. Program Transformations

In the following we discuss the 3 parts that specify a program transformation in detail.

3.1 Basic Block Analysis

BMW relies on a user-defined basic block analysis in order to specify the locations within a Java method that shall be instrumented. Using aspect-oriented programming (AOP) terminology, the custom basic block analysis defines a pointcut, i.e., a set of join points. The basic block analysis implements the `BBA` interface (see Figure 5) and computes a control flow graph. Figure 5 shows part of the `CFG` interface, which represents a control flow graph as a set of basic blocks of type `BB`. BMW provides default implementations of the `CFG` and `BB` interfaces, as well as two different `BBA` implementations, `DefaultBBA` and `PreciseBBA`. For many purposes, these default implementations are sufficient, though the user may customize the default implementations or provide a completely different one.

In the `DefaultBBA` implementation, only bytecodes that may change the control flow non-sequentially (i.e., jumps, branches, return of method or JVM subroutine, exception throwing) end a basic block. Method or JVM subroutine invocations do not end basic blocks, because we assume that the execution will return after the call. This definition of basic block corresponds to the one used in [8] and is related to the *factored control flow graph* [12]. The advantage of `DefaultBBA` is that it creates rather large basic blocks. Therefore, the number of locations where instrumentation code has to be inserted is reduced, which usually helps keeping overheads (code size and execution time) low. As long as no exceptions are thrown, bytecode metrics counted per basic block are accurate. However, exceptions (e.g., an invoked method may terminate abnormally throwing an exception) may cause some imprecisions, as all bytecodes within a basic block are counted, even though some of them may not be executed in case of an exception.

The `PreciseBBA` implementation avoids this potential imprecision. It ends a basic block after each bytecode that either may change the control flow non-sequentially (as before), or may throw an exception. As there are many bytecodes that may throw an exception (e.g., `NullPointerException` may be raised by most bytecodes that require an object reference), the resulting average basic block size is smaller. Usually, this causes higher overhead for computing bytecode metrics.

3.2 Basic Block Metrics

Dynamic bytecode metrics for a program execution can be computed by aggregating the bytecode metrics of all basic blocks that are executed. Hence, for each basic block, BMW statically computes user-defined bytecode metrics that serve to parametrize the instrumentation of the given basic block. To this end, the user has to provide an implementation of `BBMetrics` (see Figure 6); `getMetrics(BB)` returns the custom bytecode metrics for a given basic block as an array of integers. As the array represents bytecode metrics of only a single basic block, the restricted range of an integer is not a problem. Fractions can be represented by two entries in the array. Internally, the program instrumentation templates may maintain bytecode counters of any type (e.g., `long` or `double`).

Counting the number of executed bytecodes is a simple but frequent use case [5, 6, 7, 18]. It can be achieved by incrementing a counter[2] in the beginning of each basic block by the number of bytecodes inside it. In this case, `getMetrics(BB)` may return an array with a single element that contains the number of bytecodes

[2] For performance reasons, the counter should be thread-local (to avoid expensive synchronization) and passed as an extra argument (because access to the counter is very frequent) [8, 18].

in the given basic block. The examples in Sections 4.1 and 4.2 are based on this simple bytecode metric.

In a more complex setting, getMetrics(BB) may compute the number of bytecodes of certain classes (e.g., integer arithmetic, floating point arithmetic, stack manipulation, etc.), where each element position in the returned integer array corresponds to a particular bytecode class.

Another possibility is to multiply each bytecode with a well-chosen weight related to the complexity of the instruction. Weighted bytecode metrics may serve to estimate other metrics (e.g., CPU time), which can be valuable in cross-profiling use cases [33].

In order to compute control flow metrics, the custom basic block analysis may assign a unique identifier (an integer value) to each basic block, which can be preserved in the array returned by getMetrics(BB). We will sketch such a use case in Section 4.3.

3.3 Program Transformation Templates

Program transformations are defined as code templates, implementing the Templates interface shown in Figure 7. The method getExtraArgTypes() returns an array representing the types of extra arguments to be added to all Java methods. Introducing extra arguments is a convenient and efficient way of passing thread-local, possibly calling context-sensitive data structures to each Java method. If an extra argument is accessed frequently, a just-in-time compiler may allocate it to a processor register. We use the JVM encoding of field types (see [21], Section 4.3.2). E.g., if there are 2 extra arguments, one integer and one hashmap, getExtraArgTypes() would return {"I", "Ljava/util/HashMap;"}.

The methods onMethodEntry(MID), onHandlerEntry(MID), onBBEntry(MID, int[]), and onWrapper(MID), which we will call *template methods* in the following, define the program transformation templates. BMW partially evaluates these template methods and inserts the resulting code in the appropriate locations of each Java method. The partially evaluated code of onMethodEntry(MID), onHandlerEntry(MID), and onBBEntry(MID, int[]) is inserted in the beginning of a Java method, in the beginning of each exception handler, and in the beginning of each basic block. onWrapper(MID) is needed to generate wrapper methods for compatibility with native code. The template methods are not allowed to access the this reference and cannot make use of inheritance.[3]

BMW is able to generate a method identifier for each Java method, which is important for program transformations that generate calling context-sensitive data structures, such as the Calling Context Tree [1]. These data structures may store references to method identifiers to distinguish calling contexts. Method identifiers are of type MID (see Figure 7). They are allocated at runtime of a transformed program in the static (i.e., class-wide) initializers and stored in static fields. If the template methods make use of method identifiers, BMW generates the necessary static fields and generates code for the static initializers. The user may provide a factory at runtime (as a system property) in order to allocate customized method identifiers. If no factory is specified, a default implementation of MID is used, which preserves class name, method name, and method signature. Details on MID implementations and factories are omitted here, because they do not affect the program transformations.

All four template methods take an argument of type MID, which is treated specially. This is a read-only argument, as the template

[3] BMW verifies that all constraints on template methods are respected before transforming any input classes.

```
// allows template methods to access the extra arguments
// introduced by the transformation
public final class ExtraArgs {
  public static Object loadObj(int idx) { return null; }
  public static void storeObj(Object value, int idx) {}

  public static int loadInt(int idx) { return 0; }
  public static void storeInt(int value, int idx) {}

  // load and store methods for all other basic types
  // (byte, short, long, char, boolean, float, double)
  ...
}
```

Figure 8. ExtraArgs class

methods are not allowed to change the reference. BMW replaces any access to the local variable corresponding to the MID argument with a getstatic JVM instruction that loads the MID instance of the method the template currently is being applied to.

As second argument, onBBEntry(MID, int[]) takes an integer array, which is the result of invoking the user-defined BBMetrics implementation for the current basic block. The implementation of onBBEntry(MID, int[]) may only read, not modify the elements of the integer array. The array indices must be constants. BMW replaces these array accesses with the integer constants that were previously computed for the basic block. Such replacements may result in dead code, which BMW is able to remove. E.g., consider the case of counting the number of executed bytecodes for different classes of bytecode instruction (e.g., stack manipulation, field access, method invocation, etc.). Each class of bytecode instruction corresponds to a position in the integer array. As a basic block may not have bytecodes of each class, elements of the integer array may be zero. Hence, an implementation of onBBEntry(MID, int[]) may include conditionals that check whether an element of the array is not zero before updating some thread-local counters. Such code patterns can be easily identified by BMW as dead code, and hence removed.

All four template methods may use the static load and store methods defined in the ExtraArgs class (see Figure 8) in order to access the extra arguments. There are separate load and store methods for each Java basic type and for object references. The argument int idx identifies the index of the extra argument to access; the type of that argument is defined by getExtraArgTypes()[idx]. The argument int idx must be a constant, which will be compiled to a JVM instruction that pushes an integer constant onto the stack. BMW replaces the invocations of these static methods with JVM instructions that access the JVM local variables the extra arguments have been mapped to. Note that the static methods defined in ExtraArgs are never executed at runtime, so their implementation does not matter and we just provide the most trivial method bodies.

Algorithm 1 explains the steps performed by BMW to transform a given class **C**. Several steps involve the partial evaluation of template methods, as explained in Algorithm 2.

4. Use Cases

In this section we describe several use cases for BMW, including resource management (Section 4.1) and profiling (Section 4.2). Moreover, we highlight the flexibility of BMW, which is also able to generate control flow metrics (Section 4.3).

4.1 Resource Management

The idea of exploiting the number of executed bytecodes for resource management purposes, such as limiting the resource consumption of untrusted mobile code (resource control) or monitoring

Algorithm 1 Transformation of class **C**

```
1. Let createMIDs be true if at least one of the 4 template methods refers
   to the method identifier (the MID argument).
2. For each Java method m in C:
   (a) If createMIDs is true, create a static field in C to hold the method
       identifier (type MID) for m and extend the static initializer of C to
       allocate and store the method identifier.
   (b) Let cfg be the control flow graph of m, computed by the user-
       defined BBA implementation.
   (c) For each basic block bb in cfg:
       • Let integer array metrics(bb) be the bytecode metrics for bb,
         statically computed by the user-defined BBMetrics implemen-
         tation.
   (d) Extend method signature of m. The types of the extra arguments
       are provided by the user-defined getExtraArgTypes() method.
   (e) Update JVM local variable indices in m so that the local variables
       corresponding to the extra arguments are not used. Only the par-
       tially evaluated template methods will refer to these local variables.
   (f) Update method invocations in m in order to pass the extra argument.
       Note that method invocations within template methods are not
       updated.
   (g) For each basic block bb in cfg:
       • Insert partially evaluated template method
         onBBEntry(MID, int[]) in beginning of bb. For the
         partial evaluation, metrics(bb) is used as the second argument
         of onBBEntry(MID, int[]).
   (h) Insert partially evaluated template method onMethodEntry(MID)
       in beginning of m.
   (i) Insert partially evaluated template method
       onHandlerEntry(MID) in beginning of each exception handler
       in m.
   (j) Remove JVM nop instructions and unnecessary jumps in m, which
       have been introduced by the partial evaluation of template methods.
   (k) Generate wrapper method with unmodified signature. Insert par-
       tially evaluated template method onWrapper(MID) in begin-
       ning of the wrapper, then invoke m with the extra arguments
       and return the result of the invocation. The template method
       onWrapper(MID) has to provide the values for the extra arguments
       (ExtraArgs.storeX(value, i)).
3. For each native method n in C:
   • Create a 'reverse' wrapper for n. The 'reverse' wrapper has the
     extended signature (extra argument types are provided by the
     getExtraArgTypes() method) and simply discards the extra ar-
     guments before invoking n. 'Reverse' wrappers allow transformed
     Java methods to invoke any method with the extra arguments.
4. For each abstract method a in C:
   • Add an abstract method with the extended method signature to C.
```

Algorithm 2 Partial evaluation of the compiled template method **T** applied to method **m**

```
1. Replace JVM instructions in T that load the MID argument (e.g.,
   aload_1) with a getstatic instruction to load the method identifier
   of m.
2. If T is a template onBBEntry(MID, int[]) to be applied to the ba-
   sic block bb, replace the JVM instructions in T that load the elements
   of the int[] bbMetrics template argument with the corresponding
   constants in metrics(bb). This transformation is simplified by the re-
   striction that the array indices must be constants.
3. Remove dead code in T, which the previous step may have revealed.
4. For each extra argument $i$, compute the corresponding JVM local vari-
   able index lvar($i$). To compute lvar($i$), the signature of m and the extra
   arguments provided by getExtraArgTypes() are taken into account.
5. Update JVM local variable indices in T in order to avoid any interfer-
   ence with local variables in m as well as with local variables corre-
   sponding to the extra arguments (lvar($i$)).
6. Replace each invocation of ExtraArgs.load$X$($i$) ($i$ is a constant)
   in T with a corresponding JVM load instruction of local variable
   lvar($i$). The type of the JVM load instruction corresponds to the
   type $X$ (iload or iload_$n$ is also used for byte, short, char, and
   boolean). In order to avoid complex code analysis, we require the
   invokestatic JVM instruction to occur directly after an instruction
   to push the constant $i$ onto the stack (iconst_$n$, bipush, sipush,
   or ldc). Code generated by standard Java compilers meets this re-
   striction. ExtraArgs.loadObj(int) is treated specially: As it is fre-
   quently followed by a type check (JVM checkcast instruction), BMW
   uses the type information concerning the extra arguments (returned by
   getExtraArgTypes()) in order to remove unnecessary type checks.
7. Replace each invocation of ExtraArgs.store$X$($value$, $i$) in T
   with a corresponding JVM store instruction to local variable lvar($i$).
   The type of the JVM store instruction corresponds to the type $X$
   (istore or istore_$n$ is also used for byte, short, char, and boolean).
   As before, complex code analysis is avoided.
8. Add a JVM nop instruction at the end of T and replace each occurrence
   of the JVM return instruction in T with a jump to the final nop
   instruction. This allows T to be inserted into m.
```

the execution of components in a server environment (resource accounting), was first explored in [8]. In [7, 18] a flexible resource management framework, J-RAF2[4], was presented that builds on this idea.

While the original version of J-RAF2 was directly implemented by low-level bytecode instrumentation using BCEL [14], it may now be expressed as a BMW transformation with far fewer lines of code, as illustrated by class `RM` in Figure 9. `RM` relies on the default basic block analysis introduced in Section 3.1. It transforms programs to make them compute a single metric, the number of executed bytecodes, as a side-effect to their original behaviour. Hence, the metric statically pre-computed for a given basic block is the number of bytecodes constituting the block. Because `BBSize`, an implementation of `BBMetrics`, will also be used in Section 4.2, it is here defined as a separate class (and not as an anonymous inner class, such as the `Templates` implementation).

`RM` extends the signatures of all Java methods in order to pass one extra argument of type `Acc`, which is an account object maintaining the bytecode counter `consumption` of the current thread. This extra argument is normally passed on sequentially through invocations from method to method. However, whenever a Java method is invoked from non-transformed classes, from native code, or through reflection – such as at program startup – then the corresponding wrapper method with original signature will intercept the invocation, obtain the `Acc` instance of the current thread, and finally use this value to (re-)introduce the additional `Acc` argument into the normal flow of invocations.

At the beginning of every basic block, `consumption` is decremented by the adequate number of bytecodes, and a polling conditional checks whether `consumption` is less than or equal to zero. In this case, the transformed application will invoke `triggerConsume()`, the role of which is to enforce a dynamically selectable resource management policy [7] and, incidentally, to reset `consumption` to a positive value (the polling interval). The bytecode counter `consumption` is made to run towards zero, because in general it is more efficient to compare against zero than

[4] `http://www.jraf2.org/`

```
public class RM implements Transformation {
   public BBA getBBA() { return new DefaultBBA(); }
   public BBMetrics getBBMetrics() { return new BBSize(); }
   public Templates getTemplates() {
      return new Templates() {
         private final String[] extraArgTypes = { "LAcc;" };
         public String[] getExtraArgTypes() {
           return extraArgTypes;
         }
         public void onMethodEntry(MID m) {}
         public void onHandlerEntry(MID m) {}
         public void onBBEntry(MID m, int[] bbMetrics) {
            Acc acc = (Acc)ExtraArgs.loadObj(0);
            if ((acc.consumption -= bbMetrics[0]) <= 0)
               acc.triggerConsume();
         }
         public void onWrapper(MID m) {
            ExtraArgs.storeObj(Acc.getCurrentAcc(), 0);
         }
      };
   }
}

public class BBSize implements BBMetrics {
   // number of bytecodes in given bb
   public int[] getMetrics(BB bb) {
      int[] bbMetrics = new int[1];
      bbMetrics[0] = bb.getSize();
      return bbMetrics;
   }
}

// Acc represents an accounting object.
// Each thread has associated a separate Acc instance,
// which encapsulates the thread's bytecode counter.
// Acc is a runtime class, also needed to compile RM.
public class Acc {
   private static final ThreadLocal tl = new ThreadLocal();
   public int consumption; // bytecode counter
   public static Acc getCurrentAcc() {
      Acc acc = (Acc)tl.get();
      if (acc == null) { acc = new Acc(); tl.set(acc); }
      return acc;
   }
   public void triggerConsume() { ... }
   ...
}
```

Figure 9. Example: Transformation for resource management

any other arbitrary threshold. The type cast in the first line of `onBBEntry(MID, int[])`, which is necessary to compile the template method, is removed during the partial evaluation. Note that in this example, neither of the four template methods refers to the `MID` argument: BMW will therefore not create code that allocates method identifiers. Moreover, note that `Acc` is not needed during BMW execution (i.e., during the code transformations), but is required for compiling the template methods and for the execution of instrumented programs.

See Figure 10 for an example application of the transformation in Figure 9.[5] The original, compiled method `max(int, int, int)` (Figure 10, left) consists of a single basic block with 8 bytecode instructions since `DefaultBBA` is used. The transformation adds the extra `Acc` argument (Figure 10, right), which is directly passed to callee methods (i.e., here, to the invocations of the `max(int, int)` method).

In [18] we evaluated the overhead caused by such a transformation with the SPEC JVM 98 benchmark suite [32] on Sun JDK 1.5.0 (client and server mode) and on IBM JDK 1.4.1. The transformation was applied to all classes, including the JDK. On average, the

[5] For the sake of better readability, in this paper we show the results of applying transformations as Java code, whereas BMW works at the JVM bytecode level.

```
class X {                              class X {
  int max(int a, int b,                  int max(int a, int b,
          int c) {                               int c, Acc acc) {
                                           if ((acc.consumption -= 8) <= 0)
                                             acc.triggerConsume();
    return                                 return
      max(a, max(b, c));                     max(a, max(b, c, acc), acc);
  }                                      }

                                         // wrapper
                                         int max(int a, int b, int c) {
                                           Acc acc = Acc.getCurrentAcc();
                                           return max(a, b, c, acc);
                                         }
  ...                                    ...
}                                      }
```

Figure 10. Example: Applying transformation of Figure 9 for resource management

overhead was 20–30%. Without mapping thread-local data structures to extra arguments, the incurred overhead was significantly higher.

4.2 Sampling Profiling

In [5, 6] a program transformation scheme was presented for deterministic sampling profiling of Java programs. A user-defined profiling agent is invoked by each thread after the execution of a certain number of bytecodes, which is dynamically adjusted by the profiling agent. The profiling agent processes a dedicated representation of the calling thread's execution stack (a reified stack), which consists of an array of method identifiers and a stack pointer (index of the next free element in the array). Therefore, the profiling agent is able to generate calling context-sensitive profiles that approximate the (relative) number of bytecodes executed in each calling context. In [5, 6] the author showed that this approach reconciled high profile accuracy with low overhead.

In order to illustrate the versatility of BMW, we show a new version of this transformation scheme in Figure 11. We use the same basic block analysis algorithm and `BBMetrics` implementation as in the previous example. Three extra arguments are passed to all Java methods, the thread context `TC` (resembling the account `Acc` used in the previous example) and the reified stack, consisting of an array `MID[]` and an integer. `onMethodEntry(MID)` pushes the method identifier of the callee method onto the reified stack. Code for bytecode counting and polling is inserted in each basic block, in a way similar to the previous example. If the bytecode counter `instrCounter` is less than or equal to zero, the method `triggerSampling(MID[], int)` invokes the custom profiling agent to process a sample of the current reified stack and to reset `instrCounter` to a positive value. `onWrapper(MID)` obtains the calling thread's `TC` instance similarly to the previous example. It also allocates the array of the reified stack and initializes the stack pointer with zero.

Figure 12 shows an example application of the transformation in Figure 11. In contrast to the previous example, there is a template method (`onMethodEntry(MID)`) that accesses the method identifier. Hence, BMW creates a method identifier for each Java method, which is allocated in the static initializer. `MIDFactory` is a runtime class; the static method `createMID(Class, String)` either creates default method identifiers or delegates to a user-defined factory in order to allocate custom method identifiers.

In [5, 6] we evaluated this program transformation scheme for sampling profiling with the SPEC JVM 98 benchmark suite [32] on Sun JDK 1.5.0 (client and server mode) and on IBM JDK 1.4.2. Depending on the polling interval (500–1 000 000 bytecodes), we observed an average overhead of 43–96%. We also compared with

```
public class Sampling implements Transformation {
   public BBA getBBA() { return new DefaultBBA(); }
   public BBMetrics getBBMetrics() { return new BBSize(); }
   public Templates getTemplates() {
      return new Templates() {
         private final String[] extraArgTypes =
                                 { "LTC;", "[LMID;", "I" };
         public String[] getExtraArgTypes() {
            return extraArgTypes;
         }
         public void onMethodEntry(MID m) {
            MID[] stack = (MID[])ExtraArgs.loadObj(1);
            int sp = ExtraArgs.loadInt(2);
            stack[sp] = m;
            ExtraArgs.storeInt(sp+1, 2);
         }
         public void onHandlerEntry(MID m) {}
         public void onBBEntry(MID m, int[] bbMetrics) {
            TC tc = (TC)ExtraArgs.loadObj(0);
            if ((tc.instrCounter -= bbMetrics[0]) <= 0) {
               MID[] stack = (MID[])ExtraArgs.loadObj(1);
               int sp = ExtraArgs.loadInt(2);
               tc.triggerSampling(stack, sp);
            }
         }
         public void onWrapper(MID m) {
            ExtraArgs.storeObj(TC.getCurrentTC(), 0);
            ExtraArgs.storeObj(new MID[TC.STACKSIZE], 1);
            ExtraArgs.storeInt(0, 2);
         }
      };
   }
}

public class TC { // runtime class, needed to compile Sampling
   public static final int STACKSIZE = 1000;
   public int instrCounter; // bytecode counter
   public static TC getCurrentTC() { ... }
   public void triggerSampling(MID[] stack, int sp) { ... }
   ...
}
```

Figure 11. Example: Transformation for sampling profiling

```
class X {                    class X {
                                private static final MID mid_max =
                                  MIDFactory.createMID(
                                    Class.forName("X"),
                                    "max(III)I");

  int max(int a, int b,         int max(int a, int b,
          int c) {                      int c, TC tc,
                                        MID[] stack, int sp) {
                                  stack[sp++] = mid_max;
                                  if ((tc.instrCounter -= 8) <= 0)
                                    tc.triggerSampling(stack, sp);
     return                       return
       max(a, max(b, c));           max(a, max(b, c,
                                              tc, stack, sp),
                                        tc, stack, sp);
  }                             }

                                // wrapper
                                int max(int a, int b, int c) {
                                  TC tc = TC.getCurrentTC();
                                  MID[] stack =
                                    new MID[TC.STACKSIZE];
                                  int sp = 0;
                                  return max(a, b, c,
                                             tc, stack, sp);
                                }
  ...                           ...
}                             }
```

Figure 12. Example: Applying transformation of Figure 11 for sampling profiling

the standard 'hprof' profiling agent ('hprof' is part of many JDK distributions) in its sampling mode, which caused higher overhead and resulted in less accurate profiles [5, 6].

In addition to sampling profiling, we also used BMW to re-implement a variation of the transformation scheme for exact profiling described in [5], which computes a complete Calling Contex Tree [1] for each thread, offering calling context-sensitive method invocation and bytecode counters. Due to space limitations, we cannot present the details here.

4.3 Control Flow Metrics

While the previous example illustrated profiling on a per-method basis, BMW is also able to generate metrics that provide information concerning the control flow within methods. In this section we just explain the principles, since space limitations prevent us from including example transformation sources.

First, it is necessary to uniquely identify basic blocks. Hence, we extend the default implementation of BB to store an integer basic block identifier which is unique within the enclosing method. A customized BBA implementation assigns these identifiers and writes a mapping from ⟨class name, method name and sig., basic block identifier⟩-triples to the corresponding location in the program bytecode under analysis into a file. This mapping is needed to interpret the metrics generated by the transformed program, either at runtime (e.g., a debugger displaying the code being executed) or afterwards (e.g., to evaluate control flow-sensitive profiles after program termination).

Next, a custom BBMetrics implementation has to include the identifier of a given basic block in the integer array returned by getMetrics(BB) in order to make the identifier available to onBBEntry(MID, int[]).

Finally, onBBEntry(MID, int[]) may collect information concerning the executed basic blocks within a hashmap, where the keys can be generated from the method identifier and the basic block identifier, and the values represent the collected statistics. For performance reasons, a non-synchronized, thread-local hashmap should be used and passed as an extra argument in the methods of the instrumented program.

Note that it is also possible to keep track of execution paths, as onBBEntry(MID, int[]) may store the current basic block identifier within an extra argument. In this case, onMethodEntry(MID) and onHandlerEntry(MID) can be used to initialize the extra argument.

5. Related Work

In this section we review some related work in the following areas: *bytecode manipulation*, *aspect-oriented programming*, and *profiling*.

5.1 Bytecode Manipulation

Altering Java semantics via bytecode transformations is a well-known technique [31] and has been used for many purposes that can be generally characterized as adding reflection or aspect-orientedness to programs. When working at the bytecode level, the program source code is not needed.

There are many tools for manipulating JVM bytecode. The bytecode engineering library BCEL [14] represents method bodies as graph structures. Individual bytecode instructions are mapped to Java objects. BMW is based on BCEL, because the graph representation of method bodies eases instrumentation at the basic block level.

ASM [24] is a lightweight bytecode manipulation framework designed for dynamic load-time transformation of Java class files. While the instrumentation process using ASM may in many cases be more efficient than using BCEL, we found that BCEL gives finer control on the generated code, and that its representation of method bodies is better suited for instrumentation at the basic block level.

Javassist [9, 11] and Soot [34] are further examples of bytecode manipulation libraries implemented in Java. Javassist enables structural reflection and provides convenient source-level abstractions. Soot is a framework for analyzing and transforming JVM bytecode that offers four intermediate code representations.

5.2 Aspect-oriented Programming

Some tools for aspect-oriented programming in Java, such as AspectJ [19], work at the bytecode level as well. However, usually such tools support only higher-level pointcuts, such as method invocations, whereas our collection of bytecode metrics requires transformations at the basic block level. Furthermore, the extension of method signatures, which we found essential to efficiently compute thread-local, calling context-sensitive profiling data, is usually not supported by aspect-oriented programming tools. Many extensions to AspectJ have been proposed in the literature, such as e.g. new types of pointcuts [22, 26].

Frameworks such as Josh [10] or abc [3] ease the implementation of such extensions. Josh [10] is an AspectJ-like language based on Javassist [9], which allows to define new pointcut designators. abc [3] is an extensible AspectJ compiler. Its frontend is based on Polyglot [23], an extensible compiler framework for Java, whereas the backend relies on the Soot [34] framework. It certainly is possible to use such tools to implement instrumentation extensions to AspectJ for bytecode metrics computation; it would however require more development effort than with our approach based on instrumentation templates. In contrast to general-purpose tools for aspect-oriented programming, BMW is a lightweight approach specialized for the kind of instrumentation that is needed to efficiently compute dynamic bytecode metrics.

5.3 Dynamic Metrics and Profiling

In [15] the authors present a variety of dynamic metrics, including bytecode metrics, for selected Java programs, such as the SPEC JVM 98 benchmarks [32]. They introduce a tool called *J [16] for the metrics computation. *J relies on the JVMPI [20, 28], a former profiling interface for the JVM, which is known to cause very high measurement overhead[6] and requires profiling agents to be written in native code, contradicting the Java motto 'write once, run anywhere'. Because of the high overhead, tools like *J may only be applied to programs with a short execution time. In contrast, BMW-based bytecode metrics computation minimizes the overhead and can be implemented in pure Java. Therefore, it is possible to instrument long-running, complex applications in a way that is portable across different virtual execution environments.

There is a large body of related work in the area of profiling. Fine-grained instrumentation of binary code has been used for profiling by Ball and Larus [4]. The ATOM framework [27] has been successfully used for many profiling tools that instrument binary code. However, as binary code instrumentation is inherently platform-dependent, this technique is not appropriate to build tools for the platform-independent performance analysis of software components.

Whereas BMW-based profiling tools are rather intended for use at development time, sampling-based profiling is often employed in already deployed systems as support for feedback-directed optimizations in dynamic compilers [2]; indeed, sampling-based profilers may yield a sufficiently low overhead to become usable at production time. The framework presented in [2] 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 high accuracy and low overhead, as execution proceeds most of the time inside the lightly instrumented code portions. Moreover, in this setting instruction counting causes only minimal overhead, since it is implemented directly within the JVM, whereas in our approach, the instruction counting is part of the bytecode, and contributes significantly to the experienced overhead, the rest of the instrumentation representing a fairly small proportion of the overhead. Therefore, the code duplication scheme of reference [2] would make much less sense with BMW.

Hardware performance counters that record events, such as executed instructions, elapsed cycles, pipeline stalls, cache misses, etc., are often exploited for profiling. In [1] hardware performance metrics are associated with execution paths. The Jikes RVM[7], an open source research virtual machine, has been enhanced to generate traces of hardware performance monitor values for each thread in the system [30]. In [17] the authors introduce 'vertical profiling', which combines hardware and software performance monitors in order to improve the understanding of system behaviour by correlating profile information from different levels. All these approaches aim at generating precise profiling information for a particular environment, with a focus on improving virtual machine implementations. While BMW has been primarily designed to generate profilers that compute dynamic bytecode metrics in a platform-independent way, it is also possible to write transformation templates that exploit platform-specific features. For instance, we already implemented a BMW-based profiling tool that obtains information from hardware performance counters using the Performance Counter Library PCL[8].

BMW can be regarded as a profiler generator. Hence, it is related to ProfBuilder [13], a toolkit for building Java profilers. However, ProfBuilder does not address issues concerning multithreading and native code, and the generated profiling tools described in [13] cause high overhead. The authors of ProfBuilder show with a case study that profiles based on bytecode metrics are valuable to detect algorithmic inefficiencies and help the developer focus on those parts of a program that suffer from high algorithmic complexity.

Much of the know-how worked into BMW comes from our previous experience gained with program transformations for resource management [7, 8, 18] and profiling [5, 6]. The development of these transformation schemes provided us with much insight concerning the features that BMW had to support in order to reconcile ease of use, flexibility, and low overhead due to the instrumentation.

6. Discussion

In the following we discuss the strengths and limitations of our approach and give an overview of our ongoing research to further improve BMW.

6.1 BMW Benefits

Our primary goal was to develop a generic tool that allows to quickly implement bytecode instrumentation schemes in order to efficiently collect dynamic bytecode metrics. BMW fully meets this goal. Relevant bytecode instrumentation schemes can be spec-

[6] Overheads caused by profiling agents based on the JVMPI [28] or the more recent JVMTI [29] may easily exceed factor 4 000, because certain profiling events prevent runtime optimizations, such as just-in-time compilation [5].

[7] `http://jikesrvm.sourceforge.net/`

[8] `http://www.fz-juelich.de/zam/PCL/`

ified within a few lines of Java code, as illustrated in Section 4. In comparison to our previous implementation of program transformation schemes using a general-purpose bytecode manipulation library, BMW helped us to reduce the development effort for an instrumention scheme from approximately one man month to a few hours.

We designed BMW in particular to ease the computation of dynamic bytecode metrics, which has valuable use cases in the area of resource management [7] and profiling [5, 6]. We successfully re-implemented our instrumentation schemes for resource management and profiling with BMW. BMW also allowed us to easily implement further instrumentation schemes, such as the computation of dynamic control-flow metrics (see Section 4.3). Moreover, we implemented a BMW-based tool to measure the CPU time for individual basic blocks with high accuracy, exploiting processor hardware cycle counters. This information will serve us to correlate platform-independent, dynamic bytecode metrics with actual CPU time consumption on a given target system, as discussed at the end of this section.

BMW has been designed with a particular focus on implementing instrumentation schemes that efficiently compute per-thread, calling context-sensitive bytecode metrics, as illustrated by our profiling use case. Traditional approaches to compute such data rely on a modified JVM or on a standard profiling interface, such as the JVMPI [28] or the JVMTI [29]. These approaches limit portability; in the case of a modified JVM this limitation is obvious, in the case of the JVMPI or JVMTI, profiling agents have to be written in native code. Moreover, profilers based on the JVMPI or JVMTI often cause excessive overhead; e.g., we measured overheads of up to factor 4 000 for the standard `hprof` profiling agent in its exact profiling mode [5]. In contrast, BMW-based bytecode instrumentation does not prevent any optimizations performed by the Java runtime system and therefore results in moderate overhead. BMW's specific support for passing calling context-sensitive, thread-local data structures as extra arguments is crucial to achieve relatively low overhead.

In a system composed of multiple software components, it is possible to selectively instrument only certain components of interest, thus reducing the overall overhead. However, classes that may be used by different components, such as the classes of the JDK, should always be instrumented.

6.2 BMW Limitations

Concerning limitations, the major hurdle of our approach is that bytecode instrumentation does not cover the execution of native code. This is an inherent problem of BMW, since it relies on the transformation of Java code and focuses on the computation of dynamic bytecode metrics. For programs that heavily depend on native code, dynamic bytecode metrics may not be relevant.

If classes are transformed only statically before program execution, dynamically created or downloaded classes are not instrumented. One solution to this problem is to install a dedicated classloader that instruments such classes at load time.

Another solution may exploit new features introduced in JDK 1.5, which enable Java agents to instrument classes at load time. Java agents are specified with the '`-javaagent`' command-line option and rely on the instrumentation API (package `java.lang.instrument`) to install bytecode transformations. Java agents are first activated after the JVM has been initialized, i.e., before the real application. They may even redefine the already loaded system classes. However, JDK 1.5 imposes several restrictions on the redefinition of previously loaded classes. For instance, fields or methods cannot be added, and method signatures cannot be changed. Some of the features offered by BMW (e.g., passing extra arguments, provision of method identifiers) are not compatible with these restrictions. Because we want to transform all classes according to the same scheme, the current version of BMW does not use the instrumentation API.

Another limitation with BMW is that the introduction of extra method arguments may break existing code that relies on the reflection API. The methods `getConstructors()`, `getDeclaredConstructors()`, `getMethods()`, and `getDeclaredMethods()` of `java.lang.Class` return arrays of reflection objects (i.e., instances of `java.lang.reflect.Constructor` resp. `java.lang.reflect.Method`), representing wrapper methods (with the unmodified signatures) as well as methods with our extended signatures. If an application selects a method from this array considering only the method name (but not the signature), it may try to invoke a method with extended signature, but fail to provide the extra arguments, resulting in an `IllegalArgumentException`. We solve this issue by patching the aforementioned methods of `java.lang.Class` to filter out the reflection objects that represent methods with extended signatures. This modification is straightforward, because in standard JDKs these methods are implemented in Java (and not in native code).

6.3 Ongoing Research

Concerning ongoing research, we are implementing several extensions in BMW. In the current version, each basic block is instrumented in the same way by the partially evaluated method template `onBBEntry(MID, int[])`. To allow for more freedom, the next version of BMW will support multiple `onBBEntry(MID, int[])` templates that are selectively applied to different basic blocks, depending on a list of template identifiers stored there beforehand by the custom basic block analysis implementation. This approach offers the necessary flexibility to implement many optimizations (e.g., optimized instrumentation of leaf methods, inserting polling conditionals only in selected strategic locations such as in loops, etc.).

Another important improvement is the support for instrumenting dynamically created and loaded classes, which are not available at the time of static instrumentation. We started experiments with Java instrumentation agents introduced in JDK 1.5, but found it difficult to work around its restrictions on the redefinition of JDK core classes in an efficient way. Our current version for dynamic instrumentation excludes certain core classes from transformation.

More research is needed in order to assess to which extent and under which conditions bytecode metrics allow a sufficiently accurate prediction of CPU time for arbitrary programs on a given concrete system. For this purpose, individual (sequences of) bytecodes may receive different weights according to their complexity. This weighting is specific to a particular execution environment and may be generated by a calibration mechanism. As mentioned before, we already implemented a BMW-based tool to measure the CPU time for individual basic blocks with high accuracy. The collected information will be used to specify a mathematical optimization problem in order to assign bytecode weights. This approach is related to the work described in [33].

7. Conclusion

Bytecode instrumentation is a promising approach to collect dynamic metrics without causing excessive overhead. As prevailing, general-purpose, low-level bytecode engineering libraries are difficult to use and require much programming effort, there is a need for specialized toolkits that provide an easy-to-use interface in order to rapidly implement such bytecode instrumentation schemes.

In this paper we introduced an original, template-based approach to bytecode instrumentation, which has been tailored for instrumentation schemes that collect calling context-sensitive bytecode metrics. Our approach offers several advantages, such as ease of use, high flexibility and customizability, as well as dedicated support for efficient instrumentation schemes.

References

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

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

[3] 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.

[4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

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

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

[7] W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, 8(5):74–83, Sep./Oct. 2004.

[8] W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in Java. *ACM SIGPLAN Notices*, 36(11):139–155, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

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

[10] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111, New York, NY, USA, 2004. ACM Press.

[11] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. *Lecture Notes in Computer Science*, 2830:364–376, 2003.

[12] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31. ACM Press, 1999.

[13] B. F. Cooper, H. B. Lee, and B. G. Zorn. ProfBuilder: A package for rapidly building Java execution profilers. Technical Report CU-CS-853-98, University of Colorado at Boulder, Department of Computer Science, Apr. 1998.

[14] M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://jakarta.apache.org/bcel/`.

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

[16] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.

[17] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 251–269. ACM Press, 2004.

[18] J. Hulaas and W. Binder. Program transformations for portable CPU accounting and control in Java. In *Proceedings of PEPM'04 (2004 ACM SIGPLAN Symposium on Partial Evaluation & Program Manipulation)*, pages 169–177, Verona, Italy, August 24–25 2004.

[19] 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.

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

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

[22] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *First Asian Symposium on Programming Languages and Systems (APLAS-2003)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[23] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction: 12'th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, New York, NY, Apr. 2003. Springer-Verlag.

[24] ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.

[25] M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Communications of the ACM*, 46(10):24–28, Oct. 2003.

[26] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25, New York, NY, USA, 2004. ACM Press.

[27] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.

[28] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`.

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

[30] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Virtual Machine Research and Technology Symposium*, pages 57–72, 2004.

[31] E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002), USA*, volume 2487 of *LNCS*, pages 283–298, Oct. 2002.

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

[33] J. D. Turner. *A Dynamic Prediction and Monitoring Framework for Distributed Applications*. Phd thesis, Department of Computer Science, University of Warwick, UK, May 2003.

[34] 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.