Advanced Java Bytecode Instrumentation*

Walter Binder
Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
walter.binder@unisi.ch

Jarle Hulaas
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH–1015 Lausanne
Switzerland
jarle.hulaas@epfl.ch

Philippe Moret†
Faculty of Informatics
University of Lugano
CH–6900 Lugano
Switzerland
philippe.moret@unisi.ch

ABSTRACT

Bytecode instrumentation is a valuable technique for transparently enhancing virtual execution environments for purposes such as monitoring or profiling. Current approaches to bytecode instrumentation either exclude some methods from instrumentation, severely restrict the ways certain methods may be instrumented, or require the use of native code. In this paper we compare different approaches to bytecode instrumentation in Java and come up with a novel instrumentation framework that goes beyond the aforementioned limitations. We evaluate our approach with an instrumentation for profiling which generates calling context trees of various platform-independent dynamic metrics.

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, dynamic bytecode instrumentation, program transformations, dynamic metrics, profiling

1. INTRODUCTION

Altering Java semantics via bytecode instrumentation[1] is a well-known technique [16] 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. In previous work, we applied bytecode instrumentation in order to monitor and control resource consumption in standard Java Virtual Machines (JVMs) [4], and to generate calling-context-sensitive profiles for performance analysis [2, 3].

A common goal when applying bytecode instrumentation is full coverage of every bytecode that is being executed in the JVM. I.e., no method should 'escape' bytecode instrumentation, because incomplete instrumentation would result in incompleteness of the data collected by instrumentation-code. However, currently many applications of bytecode instrumentation suffer from one or both of the following two limitations: (1) Certain core classes of the JDK are excluded from instrumentation. Consequently, the bytecode executed by the methods in these classes is not tracked by instrumentation-code. (2) Instrumentation is performed only statically; the whole application (including libraries) has to be instrumented prior to execution. Hence, dynamically generated or downloaded code is not instrumented.

In this paper we present a generic framework for *dynamic* bytecode instrumentation in *pure Java* called FERRARI (Framework for Efficient Rewriting and Reification by Advanced Runtime Instrumentation), which enables the instrumentation of the *whole JDK* including all core classes, as well as the instrumentation of dynamically loaded classes *at runtime*. FERRARI has been designed to take an arbitrary user-defined instrumentation-process (conforming to the simple FERRARI API, which will be explained later in this paper) and to enhance it with support for full JDK instrumentation and dynamic instrumentation.

FERRARI uses a combination of static and dynamic instrumentation to achieve its goals. Only core classes of the JDK are instrumented statically, whereas all other classes are instrumented at runtime. Instrumenting core classes of the JDK requires special caution, since instrumentation-code must not disrupt the bootstrapping of the JVM. We solved this problem by ensuring that instrumentation-code is not executed before the bootstrapping phase is completed.

Dynamic instrumentation can perturbate measurements, particularly because any Java-based bytecode engineering library relies on classes of the JDK which themselves have been instrumented. Hence, we had to provide a mecha-

*This work was supported by the Swiss National Science Foundation.

†Part of this work was conducted while Philippe Moret was at the Ecole Polytechnique Fédérale de Lausanne (EPFL).

[1] In this paper, the term *instrumentation-process* refers to the code that performs an instrumentation, whereas the expression *instrumentation-code* denotes the extra code inserted by an instrumentation-process. *Instrumented-code* describes the code resulting from an instrumentation-process, i.e., including the original code as well as the inserted instrumentation-code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ 2007, September 5–7, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-672-1/07/0009 ...$5.00.

nism that allows to temporarily prevent the execution of instrumentation-code.

The original contribution of this paper is a novel framework for bytecode instrumentation in Java, which is implemented in pure Java, supports user-defined instrumentation-processes written in pure Java, ensures that every class gets instrumented, and supports dynamic instrumentation at runtime with minimal perturbations. Moreover, we evaluated our approach using an instrumentation-process for exact profiling, which collects calling-context-sensitive profiles using platform-independent dynamic metrics, such as the number of executed bytecodes per calling context. We also show that the overhead caused by our profiling approach can be orders of magnitude lower than using standard profiling interfaces, such as the JVMPI [14] or the more recent JVMTI [15].

This paper is structured as follows: Section 2 gives background information on bytecode instrumentation and explains the main difficulties when instrumenting the JDK. In Section 3 we describe our generic instrumentation framework, while Section 4 illustrates the APIs for custom instrumentation-processes. Section 5 discusses different approaches to calling context reification, which is important for instrumentations that collect calling-context-sensitive information (e.g., profiling or tracing). Section 6 presents our concrete use case, a tool for exact profiling that was adapted to exploit the new instrumentation framework. Section 7 discusses the strengths and limitations of our approach. Finally, Section 8 concludes this paper.

2. BYTECODE INSTRUMENTATION ISSUES

In the following we discuss different approaches to bytecode instrumentation and summarize the major difficulties that must be taken into consideration when designing an instrumentation-process.

2.1 Static versus Dynamic Instrumentation

Static bytecode instrumentation inserts all instrumentation-code before the program under instrumentation starts execution. The main advantage of this approach is that it causes less runtime overhead, as all classes are instrumented before the program is executed. Furthermore, static bytecode instrumentation may leverage any high-level bytecode engineering library (e.g., BCEL [9], ASM [12], Javassist [7], JOIE [8], etc.) without any risk of perturbating measurements; as static instrumentation is completed before the program under instrumentation starts execution, the overhead caused by the instrumentation-process is not an important issue. The major drawback of static instrumentation is that dynamically generated or loaded code is not instrumented.

Dynamic bytecode instrumentation is interleaved with the execution of the program under instrumentation; an instrumentation agent is invoked each time a class is loaded and may augment the loaded bytecode with instrumentation-code. On the one hand, this approach introduces extra overhead (mainly during program startup) and may perturbate measurements due to the runtime instrumentation-process. However, on the other hand, it ensures that all classes will be instrumented and avoids tedious bytecode instrumentation before program startup. Whereas static instrumentation is typically applied to all library classes, including those that are never used by an application, dynamic instrumentation processes only those classes that are actually being loaded. Moreover, dynamic instrumenation prevents certain mistakes, such as forgetting to instrument classes after modification and recompilation. Among these benefits of dynamic instrumentation, the guarantee that all loaded classes are instrumented carries most weight for us.

2.2 Instrumentation Support in Standard Java Environments

The JVMTI [15] offers a mechanism to instrument classes as they are being loaded. Unfortunately, the JVMTI requires instrumentation-processes to be implemented in native code, contradicting the Java motto 'write once, run anywhere'. As we aim at supporting instrumentation in pure Java, we chose not to rely on the JVMTI.

JDK 1.5 has introduced a mechanism, Java language instrumentation agents (package `java.lang.instrument`), to instrument classes as they are being loaded. Even though instrumentation agents are loaded and executed before the class containing the `main(String[])` method, these agents are loaded only after the JVM has completed bootstrapping. At that stage of the execution, already several hundred classes have been loaded but not been processed by any instrumentation agent. The JDK offers a mechanism to redefine these pre-loaded classes, which however imposes several strong limitations on redefinition, as summarized in the JDK 1.6 API documentation: 'The redefinition may change method bodies, the constant pool and attributes. The redefinition must not add, remove or rename fields or methods, change the signatures of methods, or change inheritance.' These limitations are far too restrictive for many instrumentation-processes, such as e.g. for calling context reification which requires the introduction of additional method arguments and therefore changes method signatures (see Section 5).

Our approach leverages the `java.lang.instrument` package for dynamic instrumentation, but we resort to static instrumentation for those core classes of the JDK that are initially loaded upon JVM startup, in order to avoid the mentioned severe restrictions imposed on class redefinitions.

2.3 Bootstrapping

For many applications of bytecode instrumentation, such as e.g. profiling, it is important that the instrumentation-code tracks each bytecode that is being executed in a system. This includes the bytecodes of application classes, libraries, as well as all classes of the JDK. However, instrumenting all classes of the JDK is difficult for several reasons, which we explain below.

Before any Java application can execute, the JVM has to bootstrap, which involves loading and initializing core classes of the JDK, mostly in the `java.lang` package (e.g., `Object`, `String`, `Throwable`, `Thread`, etc.). Most JVMs are very sensitive w.r.t. the order in which classes are loading during bootstrapping. If that order is changed due to instrumentation-code executing during the bootstrapping phase (i.e., because instrumentation-code may depend on certain classes that need to be loaded before the instrumentation-code can be executed), the JVM may crash.

Another problem during the bootstrapping phase is that there is no `Thread` object associated with the

thread that loads and links the initial JDK classes. If `Thread.currentThread()` was executed by instrumentation-code during the bootstrapping, it would return `null`.

For these reasons, instrumentation-code either has to be carefully crafted in order not to crash the JVM during bootstrapping, or there has to be a mechanism that skips execution of arbitrary instrumentation-code during bootstrapping. Since our goal is supporting arbitrary user-defined instrumentation-processes, we follow the second approach, inserting conditionals that skip instrumentation-code until bootstrapping is completed. Thus, the only instrumentation-code executed during bootstrapping are conditionals, which do not introduce any class dependencies. Note that Java mandates a lazy class initialization strategy [10] ensuring that classes are initialized upon first use, but not before. Therefore, instrumentation-code that is skipped during bootstrapping will not cause initialization of the classes it depends on.

An important issue is how to find out when the bootstrapping is over. When the JVM is ready to execute arbitrary user-defined code, it is also safe to allow execution of custom instrumentation-code. We define the bootstrapping phase to last until our instrumentation agent (from the JVM's perspective, this is arbitrary user-defined code) starts execution.

2.4 Native Code

Native code cannot be altered by bytecode instrumentation, since there is no corresponding bytecode representation. Hence, if we change bytecode, we have to make sure that such changes do not break any assumptions that native code makes on bytecode.

An obvious dependency of native code on bytecode is the invocation of a method (bytecode) by native code through the Java Native Interface (JNI) [13]. For example, if an instrumentation-process changes method signatures, it must introduce wrapper methods with the original signatures such that unmodifiable native code can call these wrappers. In Section 5 we will show a transformation that introduces extra method arguments in order to efficiently reify the calling context.

Dependencies of bytecode on native code require special attention, too. Native code can be called by bytecode through methods declared as `native`. If we change the signature of a native method, the corresponding function in a native code library would not match anymore the declared native method. There are two ways to avoid resp. solve this problem: The instrumentation-process could leave the signatures of native methods unchanged (eventually introducing native method wrappers with extended signatures that simply drop extra arguments before calling the unmodified native method), or it may leverage a new functionality introduced in JDK 1.6 called native method prefixing [15] that allows renaming a native method and introduction of a bytecode implementation with the name of the original native method.

Unfortunately, introducing wrapper methods for compatibility with native code does not always work. Invocations of wrapper methods constitute extra frames on the call stack and may break some native code for stack introspection. For instance, in Sun's JDKs there are certain methods that rely on a fixed invocation sequence. Examples include methods in `Class`, `ClassLoader`, `Runtime`, and `System`. These

```
void f() {
    X
    g();
    Y
}
```

(a) Before instrumentation

```
void f(ExtraArg a) {
    X_instrumented
    g(a);
    Y_instrumented
}

void f() { // wrapper
    ExtraArg a = ...; // create extra arguments
    f(a);
}
```

(b) Instrumentation using wrapping

```
void f(ExtraArg a) {
    X_instrumented
    g(a);
    Y_instrumented
}

void f() { // duplication
    ExtraArg a = ...; // create extra arguments
    X_instrumented
    g(a);
    Y_instrumented
}
```

(c) Instrumentation using code duplication

Figure 1: If an instrumentation-process introduces additional method arguments, compatibility with native code has to be ensured by wrapping or by code duplication techniques. Wrapping is not applicable for certain JDK methods that cannot tolerate the additional stack frame due to a wrapper. X and Y are arbitrary code blocks.

methods inspect the stack frame of the caller to determine whether an operation shall be permitted. If wrapper methods are added to the JDK, the additional stack frames due to the invocation of wrapper methods will violate the assumptions of the JDK programmer concerning the execution stack. The only way to avoid this problem is to make sure that there are never any extra stack frames when invoking methods that depend on a certain invocation sequence. This can be achieved by using code duplication instead of wrapping for compatibility with native code (see Figure 1).

2.5 Reflection

If an instrumentation-process introduces additional methods, these methods are visible through the reflection API. Existing code using reflection may get confused with such additional methods, in particular when the selection of a method to be invoked is solely based on its name, and not on the complete signature. In such a case, overloading methods with extra arguments may break existing code, since upon invocation arguments would be missing.

This issue can be solved by patching the meth-

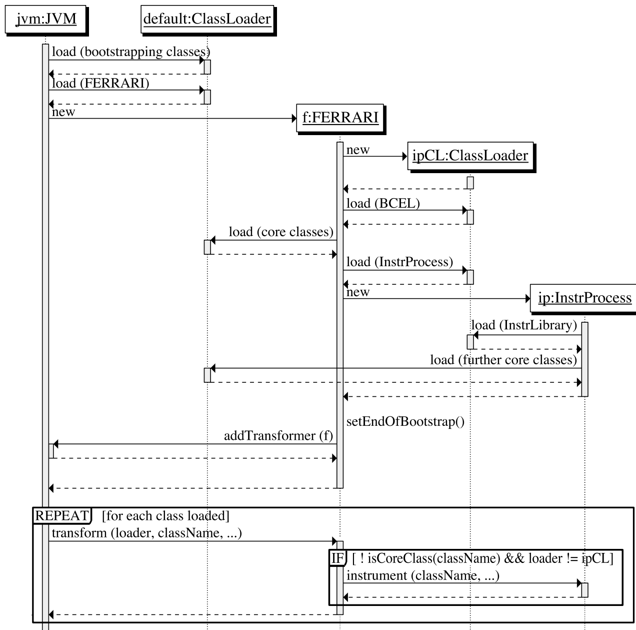

Figure 2: Simplified sequence diagram of the dynamic instrumentation environment (note that the loading of class libraries often actually is a lazy process and might thus be more realistically represented as interleavings).

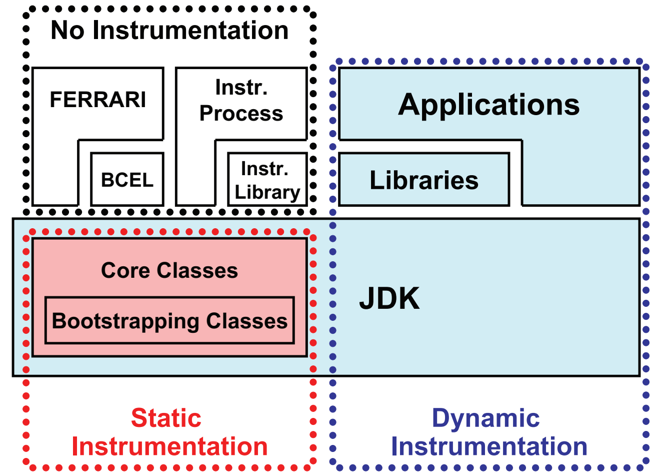

Figure 3: Overview of the runtime structure, with parts that remain uninstrumented, and other parts that are either statically or dynamically instrumented.

ods `getConstructors()`, `getDeclaredConstructors()`, `getMethods()`, and `getDeclaredMethods()` of `java.lang.Class` so as to filter out the reflection objects that represent methods that were introduced by the instrumentation-process. This modification is straightforward, because in standard JDKs the aforementioned methods of `java.lang.Class` are implemented in bytecode.

3. GENERIC INSTRUMENTATION FRAMEWORK

In the following we describe FERRARI, our generic bytecode instrumentation framework. FERRARI combines the advantages of both static and dynamic instrumentation. We statically instrument only the core classes of the JDK, which need to be loaded before dynamic instrumentation is possible, and afterwards rely on an instrumentation agent to dynamically instrument all other classes.

3.1 Core Classes and Bootstrapping Classes

FERRARI comprises two agents that rely on the package `java.lang.instrument`, a *probing agent* that gathers information regarding the set of classes to be instrumented statically, and an *instrumentation agent* for dynamic instrumentation.

We denote as C the set of *core classes* requiring static instrumentation, and as B the set of *bootstrapping classes* loaded prior to the execution of any agent. We define the *bootstrapping phase* to last until an agent executes the first bytecode; after the bootstrapping phase, arbitrary client code can execute.

The *probing agent* uses a function in package `java.lang.instrument` to compute B.[2] On a given JVM, the bootstrapping classes are always loaded before any instrumentation agent and therefore have to be instrumented statically prior to JVM startup; i.e., $B \subseteq C$. In general, $C \supset B$, since a user-defined instrumentation-process may depend on JDK core classes that are not part of B.

FERRARI's instrumentation agent uses a dedicated classloader to load the classes constituting a custom instrumentation-process and the BCEL [9] classes (i.e., the bytecode engineering library FERRARI itself depends on) into a separate namespace *IP*. Classes in *IP* are excluded from instrumentation. Depending on the custom instrumentation-process, the classes in *IP* may include further general-purpose instrumentation libraries, such as e.g. ASM [12]. Nonetheless, if such classes are also loaded by the application under instrumentation using another classloader (e.g., the system classloader), they will get instrumented as any other application class. The namespace *IP* just guarantees that the instrumentation-process is not instrumented itself.

Unfortunately, JDK classes (in package `java.*`) cannot be reloaded with a custom classloader. I.e., only a single version of a JDK class can exist within the JVM. Let D denote the set of JDK classes FERRARI and the user-defined instrumentation-process (transitively) depend on. Then, $C = B \cup D$. If D cannot be determined statically for a given instrumentation-process, the set of all JDK classes may be used as an upper bound.

After static instrumentation of the core classes C, the instrumented core classes are included in the beginning of the bootclasspath so as to replace the non-instrumented versions of these classes. After the bootstrapping phase, FERRARI's instrumentation agent dynamically instruments all subsequently loaded classes according to the user-defined instrumentation-process (with the exception of the core classes in C and classes in *IP*). To this end, it first registers

[2] `Instrumentation.getAllLoadedClasses()`

```
public class BootstrapLock {
   public static boolean isBootstrap = true;
   public static synchronized boolean isBootstrap() {
      return isBootstrap;
   }
   public static synchronized void setEndOfBootstrap() {
      isBootstrap = false;
   }
}
```

Figure 4: The `BootstrapLock` class.

a `ClassFileTransformer`, which exposes a transformation method that is subsequently invoked by the JVM whenever a new class is defined. Note that for each JVM version and each custom instrumentation-process, the core classes have to be determined (by the probing agent) and instrumented statically.

Figure 2 illustrates the dynamics of instrumentation at runtime, whereas Figure 3 provides a static view summarizing the parts of a running Java system that are excluded from instrumentation, that are instrumented statically before program execution, or that are instrumented dynamically at runtime.

3.2 Bootstrapping Flag

Concerning instrumentation of the bootstrapping classes B, it is essential not to disrupt the bootstrapping of the JVM. As FERRARI cannot determine whether a user-defined instrumentation-process would disrupt the bootstrapping, it ensures that no user-defined instrumentation-code is executed during the bootstrapping phase.

In order to disable execution of instrumentation-code during the bootstrapping phase, we use a global flag to indicate bootstrapping (see Figure 4). Initially, the flag is set; it is cleared by FERRARI's instrumentation agent (`BootstrapLock.setEndOfBootstrap()`) and remains in cleared state until JVM termination.

In a multi-threaded system, such as the JVM, it is necessary to access the bootstrap flag in a critical section (synchronized access). In order to reduce the synchronization overhead, we perform a well-known optimization, double-checked locking, which avoids the synchronization overhead once the bootstrapping phase is over and all threads have seen the cleared flag. Note that reading the static field `BootstrapLock.isBootstrap` may return the out-dated value `true`; in that case, the synchronized method `BootstrapLock.isBootstrap()` will be called, ensuring that the calling thread will read the current state of the flag, according to the Java Memory Model [10].[3]

3.3 Static Initializers

For each class, the JVM ensures that the static initializer (also known as class initializer) is executed exactly once before the class is used for the first time (e.g., before the first invocation of a static method, before the first access to a static field, or before the first object instantiation). This feature is known as Java's lazy class initialization strategy, i.e., classes are initialized upon first use, but not before [10].

Allowing instrumentation of static initializers is problematic, because bootstrapping classes must not execute any instrumentation code before the end of the bootstrapping phase. If a class is initialized during the bootstrapping phase, there is no way to re-run the instrumented static initializer after the bootstrapping phase. However, some user-defined instrumentation-processes require the insertion of static fields into all classes (e.g., the instrumentation-process for exact profiling outlined in Section 6 relies on the insertion of static fields), which may need to be initialized by the static initializers.

FERRARI solves this problem as follows: Although classes initialized during the bootstrapping phase execute the non-instrumented static initializer, the custom instrumentation-process may introduce additional classes for each instrumented class. These extra classes may include static fields and their own static initializers. As the added classes are referenced only by instrumentation-code, which is guaranteed not to execute during the bootstrapping phase, the JVM's lazy class initialization strategy ensures that the extra classes will be initialized after the bootstrapping phase, when the execution of instrumented static initializers does not cause any problem.

FERRARI ensures that added classes are placed in the same package as the class they conceptually belong to. Thus, static fields in added classes may be `public`, `protected`, or package-visible, but (typically) cannot be `private` (unless corresponding accessor methods are defined).

3.4 Constructors

Because instrumentation-processes may also insert instance fields (as opposed to static fields) to be initialized by constructors, objects created during the bootstrapping phase may be incompletely initialized, as only non-instrumented constructors are executed during the bootstrapping phase. One approach to deal with this issue is to collect all incompletely initialized objects in a structure (e.g., in a dynamically growing object array) and to pass that structure to the instrumentation-process upon end of the bootstrapping phase in order to complete initialization. However, since the instrumentation-processes we envision do not require this functionality, FERRARI currently does not support this feature. Some of our instrumentation-processes indeed insert instance fields in certain classes (e.g., in `java.lang.Thread`), but leave these fields uninitialized. Instrumentation-code is responsible for lazily initializing these fields on demand. Because added instance fields are not initialized by the constructor, they (typically) cannot be declared as `final`.

3.5 Preventing Measurement Perturbations

In order to prevent measurement perturbations, FERRARI provides mechanisms to temporarily disable the execution of instrumentation-code for each thread. This feature is important to ensure that the dynamic instrumentation-process does not cause artifacts in data structures created by instrumentation-code, such as in profiles.[4] Moreover, instrumentation-code may need to invoke non-instrumented methods or constructors as well. E.g., profiling code may

[3] Alternatively, we could have defined the flag as `volatile` in order to avoid reading an outdated value. However, `volatile` variables cause some extra overhead, whereas our approach minimizes overhead for the common case, i.e., for access after the bootstrapping phase is over.

[4] Although the classes corresponding to the custom instrumentation-process are loaded in a separate namespace and are excluded from instrumentation, they may rely on instrumented core classes.

need to allocate objects to represent profiling data; such allocations must not invoke any instrumented constructor.

FERRARI introduces a thread-local flag `execInstrCode` in order to select for each thread whether it should execute instrumentation-code. In particular, FERRARI's instrumentation agent disables this flag for the current thread before invoking a user-defined instrumentation-process. By default, FERRARI stores this flag as an instance field added to `java.lang.Thread`, although it is also possible to embed the flag in thread-local state that is reified as an extra method argument by the instrumentation-process. The latter approach often causes less overhead, since it avoids the otherwise needed calls to `Thread.currentThread()`.

During the bootstrapping phase, the `execInstrCode` flag is disabled for each thread. FERRARI's instrumentation agent is in charge of enabling the flag for each thread in the system when signaling the end of the bootstrapping phase. FERRARI instruments thread creation so that a new thread 'inherits' the flag value from the creating thread.

Because the constructor of `java.lang.Object` is often invoked by instrumentation-code, FERRARI offers a second mechanism to prevent execution of instrumentation-code. FERRARI introduces a second constructor in `java.lang.Object`, which takes an argument of type `org.ferrari.NoInstrCode` and behaves as the original, non-instrumented constructor. The argument is necessary only to distinguish the constructor signature; typically, instrumentation-code invokes the special constructor with a `null` argument. For performance reasons, FERRARI applies the same idea to several other methods which are frequently invoked by instrumentation-code, such as `Thread.currentThread()` and `System.identityHashCode(Object)`.

Please note that our approach does not prevent perturbations concerning the measurement of low-level resource consumption. Obviously, dynamic instrumentation consumes CPU and memory resources at runtime. Nonetheless, our approach ensures that instrumentation-code is not aware of dynamic instrumentation; there are no artifacts in data structures created by instrumentation-code. If an instrumentation-process collects only platform-independent dynamic metrics based on the bytecodes being executed, then our approach reduces perturbations to a minimum. In this case, the only perturbations are due to differences in thread scheduling for multi-threaded applications. In Section 6 we will refer to an instrumentation-process for exact profiling that focuses on dynamic bytecode metrics.

3.6 Instrumentation Scheme

A user-defined instrumentation may (1) modify method bodies (but leave the signatures unchanged), (2) leave methods (signature and body) unchanged, or (3) introduce new methods (with signatures that do not exist in the original program).

For case (1), Figure 5 and Figure 6 illustrate FERRARI's instrumentation scheme.[5] Figure 5 applies to bootstrapping classes, whereas Figure 6 corresponds to all other classes (whether instrumented statically or dynamically). FERRARI keeps the non-instrumented bytecode version together with the instrumented-code and inserts a conditional to select the version to be executed. Note that in the common case, reading the `isBootstrap` flag returns `false` and the static synchronized method `isBootstrap()` will not be invoked.

[5]In this paper, all instrumentation is presented at the level of Java language expressions, whereas our implementation operates at the bytecode level.

```
f() {
   if (Thread.currentThread().execInstrCode &&
       (!BootstrapLock.isBootstrap ||
        !BootstrapLock.isBootstrap())) {
      // f': method body instrumented by custom
      //     instrumentation-process
      ...
   } else {
      // f: original, non-instrumented method body
      ...
   }
}
```

Figure 5: Instrumentation scheme for bootstrapping classes.

```
f() {
   if (Thread.currentThread().execInstrCode) {
      // f': method body instrumented by custom
      //     instrumentation-process
      ...
   } else {
      // f: original, non-instrumented method body
      ...
   }
}
```

Figure 6: Instrumentation scheme for non-bootstrapping classes.

Regarding case (2), methods that are not touched by the custom instrumentation-process are left unchanged.

In case (3), methods added by the instrumentation-process are directly inserted into the final bytecode without further instrumentation. We require that such methods may only override an added method in a supertype, but not a method that already existed in the original program. This restriction guarantees that added methods can only be invoked by instrumentation-code.

FERRARI ensures that added methods are not visible through the reflection API by filtering the result returned by reflection methods in `java.lang.Class`. For this purpose, FERRARI keeps an exclusion list of methods that should be hidden from reflection. These constraints ensure that added methods are guaranteed not to execute during the bootstrapping phase and that they do not break code using reflection.

4. APIS FOR CUSTOM INSTRUMENTATION-PROCESSES

Our framework defines two APIs, a general-purpose API as well as a specialized one. Each user-defined instrumentation-process has to implement one of these APIs.

4.1 General-Purpose API

FERRARI uses the `Instrumentation` interface (see Figure 7) in order to invoke a user-defined instrumentation-process. For each class to be instrumented, FERRARI invokes the `instrument(String, byte[], Set<String>)` method of the instrumentation-process. The first two arguments correspond to the fully qualified classname and to the class bytes of the class to be instrumented. The classname is also encoded in the class bytes, but making it explicit as an argument allows the custom instrumentation-process to skip

```
public interface Instrumentation {
   Instrumented instrument(String className,
                           byte[] classBytes,
                           Set<String> nonWrappableMethods);
}

public interface Instrumented {
   Set<MethodDesc> changedMethods();
   byte[] instrumentedClass();
   Map<String, byte[]> addedClasses();
}

public interface MethodDesc {
   String getName();
   String getSignature();
   boolean isStatic();
   ...
}
```

Figure 7: General-purpose API. The custom instrumentation-process has to implement the `Instrumentation` interface, while FERRARI provides default implementations of the interfaces `Instrumented` and `MethodDesc`.

certain classes without having to analyze the class bytes.

The third argument to `instrument(String, byte[], Set<String>)` is a set with the fully qualified names of all methods in the system that must not be wrapped by the user-defined instrumentation process, in order to avoid the problem with stack introspection outlined in Section 2.4. If the instrumentation-process makes use of method wrapping, it has to check for each method whether it is included in that set. In that case, wrapping has to be replaced with code duplication techniques. The set `nonWrappableMethods` is statically defined for each JVM. We have manually analyzed Sun's JDKs 1.5 and 1.6 and provide the corresponding sets of non-wrappable methods for these platforms. For JDKs where that set is unknown, FERRARI will pass a `null` value to the instrumentation-process. In that case, the instrumentation-process should avoid wrapping any methods of the JDK.

The method `instrument(String, byte[], Set<String>)` has to return an object implementing the `Instrumented` interface, which allows FERRARI to access the following information:

- `changedMethods()` – Returns the set of methods that have been modified by the instrumentation-process. A `MethodDesc` instance uniquely identifies a method by its name, signature, etc. As the instrumentation-process is required to explicitly state the set of modified methods, FERRARI does not have to perform any complex analysis.

- `instrumentedClass()` – Returns the class processed by the custom instrumentation-process as a byte array.

- `addedClasses()` – Allows to add extra classes (see Section 3.3). The added classes are returned as a `Map` of strings to byte arrays, which correspond respectively to the classnames and bytes of the added classes. The extra classes are added to the package of the instrumented class.

4.2 Optimized API

In the aforementioned general-purpose API, classes are passed as byte arrays between FERRARI and the user-defined instrumentation-process. On the one hand, this approach has the advantage that the developer of the instrumentation-process is free to use the bytecode instrumentation libarary of his preference, since the FERRARI API does not depend on any particular library. On the other hand, this approach may cause significant overhead, since both FERRARI and the instrumentation-process need to parse the class bytes. As dynamic instrumentation increases program execution time, particularly during program startup when most classes are loaded, it is important to avoid repeated parsing of the same class bytes.

```
public interface InstrumentationBCEL {
   InstrumentedBCEL instrument(JavaClass jcl,
                               Set<String> nonWrappableMethods);
}

public interface InstrumentedBCEL {
   Set<MethodDesc> changedMethods();
   JavaClass instrumentedClass();
   Map<String, byte[]> addedClasses();
}
```

Figure 8: Optimized API for BCEL-based instrumentation-processes.

FERRARI relies on BCEL [9] to implement the instrumentation explained in Section 3.6. When FERRARI instruments a class X at runtime (i.e., X is not a core class of the JDK), it first passes the class bytes to the instrumentation-process using the `instrument(String, byte[])` method and receives back the instrumented class bytes via the `instrumentedClass()` method. In general, FERRARI needs to parse both the original class bytes as well as the class bytes returned by the instrumentation-process, in order to emit the final class bytes. The custom instrumentation-process, too, parses the original class bytes and emits instrumented-code. Consequently, in total, instrumentation of a single class requires parsing class bytes three times and generating class bytes twice.

If we assume that the user-defined instrumentation-process uses BCEL as instrumentation library, just as FERRARI does, we can avoid repeated parsing and emitting of class bytes. Hence, FERRARI supports a second API where instances of `JavaClass` (the BCEL abstraction that represents a parsed class) are exchanged with the instrumentation-process instead of byte arrays. Figure 8 illustrates the API optimized for BCEL-based instrumentation-processes.

In contrast to the non-optimized API, the `instrument(JavaClass)` method does not take the classname as argument, because it can be easily obtained from the passed `JavaClass` instance. Classes added by the custom instrumentation-process are still represented as byte arrays, since they are not further processed by FERRARI.

FERRARI uses reflection to inspect the user-defined instrumentation-process; if it implements the `InstrumentationBCEL` interface, the optimized API is used, otherwise the previously described API. Because FERRARI needs the original class representation in addition to the user-instrumented-code in order to emit the final class bytes, FERRARI keeps a deep copy of the `JavaClass` instance that is passed to the `instrument(JavaClass)` method. The optimized API ensures that dynamic instrumentation requires parsing and emitting class bytes only once.

5. CALLING CONTEXT REIFICATION

In many applications of bytecode instrumentation, such as in profiling or trace generation, it is important to access the calling context with instrumentation-code.

In Java, the calling context is not well exposed to the programmer. The only standard API that could be used for this purpose is the `Throwable` class: Using '`new Throwable().getStackTrace()`', a thread can obtain a trace of its call stack. Unfortunately, the stack trace provides only class and method name for each stack frame, but not the method signature. As the stack trace may also include the name of the Java source file and a line number for each stack frame, it may be feasible to obtain the method signature from the source file. However, the source file may not always be available (e.g., consider the use of libraries that are distributed only as archives of compiled classes). I.e., in general it may not be possible to distinguish overloaded methods (methods of the same class with the same name but different signatures). Moreover, stack traces may be incomplete on certain JVMs, since the `Throwable` specification gives a lot of flexibility to the JVM implementor. Furthermore, if information on the current calling context is needed frequently, an approach that allocates a new `Throwable` instance each time would cause excessive overhead.

An alternative approach is to reify the calling context, i.e., to add instrumentation-code that maintains a representation of the current calling context. For instance, the calling context could be represented as a node within a Calling Context Tree (CCT) [1], or as a pair consisting of an array of method identifiers (representing the reified call stack) and an integer representing the reified stack pointer. The CCT representation is well suited for exact profiling (i.e., collecting dynamic metrics for each calling context) [2], whereas the latter approach is more appropriate for sampling profiling [3].

A naive solution is to keep the calling context of each thread in a thread-local variable and to update it upon each method entry and exit. As methods may return normally or exit abnormally throwing an exception, such an approach requires the insertion of `finally{}` clauses to restore the caller's context. However, frequent access to thread-local variables causes significant overhead on many JVMs (thread-local variables are typically implemented as a map; accessing them requires first obtaining a reference to the current thread, and second a lookup in a map), and the overhead due to a `finally{}` clause in each method can be excessive. Moreover, this approach does not track the invocation of native methods.

We found that on recent JVMs, the most efficient portable way to reify calling context is to introduce extra method arguments that represent the current calling context. E.g., in the case of a CCT representation, the caller passes its CCT node to the callee, and the callee gets or creates the child node representing the callee's context. Using additional arguments to reify the calling context avoids the problem of updating a thread-local variable upon method entry and exit. The insertion of `finally{}` clauses is not necessary.

For compatibility with native native code, it is essential to keep methods with the unmodified signature. We can use either wrapping or code duplication techniques. Typically, wrapping is preferred, since it avoids code bloat. However, there are cases where wrapping is not applicable, as explained before in Section 2.4.

The question remains how the method with the unmodified signature obtains the current reified calling context. The simplest approach is to take the root calling context, which effectively means that all Java methods invoked from native code become children of the root calling context. Depending on the concrete setting, this may be acceptable or not.

If such an inaccuracy in the reified calling context is not tolerable, we can leverage the JVMTI native method prefixing functionality (since JDK 1.6) [15] in order to instrument native methods so as to preserve the reified calling context in a thread-local variable during execution of the native method. (Upon termination of the called native method, the previously stored reified calling context has to be recovered.) Now methods with the unmodified signature (that are only invoked by native code) can take the reified calling context from the thread-local variable, instead of simply taking the root context. This approach works well with reflection, since the invocation of a method by reflection involves native code. Note that in this approach, the reified calling context is passed as extra arguments whenever possible, whereas (possibly expensive) access to the thread-local variable is needed only upon transitions between native code and bytecode.

6. CASE STUDY: EXACT PROFILING

Profiling allows a detailed analysis of the resource consumption of programs. It helps detecting hot spots and performance bottlenecks, guiding the developer in which parts of a program optimizations may pay off. Profiling provides detailed execution statistics on the basis of individual methods (e.g., call stack, invocation counter, CPU time, etc.).

A classical approach in Java [10] is to use the JVMTI [15] or JVMPI [14] interfaces which are provided by the JDK. This approach has two shortcomings: the profiling agent is written in native code and is therefore not portable; secondly, it may introduce excessive overhead, especially when used for exact profiling. For these reasons, we developed JP, a Java profiler that relies on neither of these APIs, but directly instruments the bytecode of Java programs in order to generate a CCT [1] of platform-independent dynamic metrics, such as the number of method invocations and the number of executed bytecodes for each calling context. JP relies on the insertion of extra method arguments for calling context reification (see Section 5). JP was first presented in [2], but without the dynamic instrumentation facilities and systematic coverage of all JDK classes brought by FERRARI.

Adapting JP to be usable as instrumentation-process in conjunction with FERRARI required only two minor changes to JP: (1) JP had to implement the `Instrumentation` or `InstrumentationBCEL` interface presented in Section 4, and, (2) static fields that JP inserts to hold method identifiers (immutable objects that uniquely identify the methods defined in a class) had to be moved to separate classes (see Section 3.3).

To evaluate the overhead caused by our profiling scheme, we used the SPEC JVM98 benchmark suite [18] (problem size 100), which consists of 7 benchmarks, as well as the SPEC JBB2005 benchmark [17] (warehouse sequence 1, 2, 3, 4, 5, 6, 7, 8) on a Linux Fedora Core 2 computer (Intel Pentium 4, 2.66 GHz, 1024 MB RAM). The metric used by SPEC JVM98 is the execution time in seconds, whereas SPEC JBB2005 measures the throughput in operations/second. All benchmarks were run in single-user mode

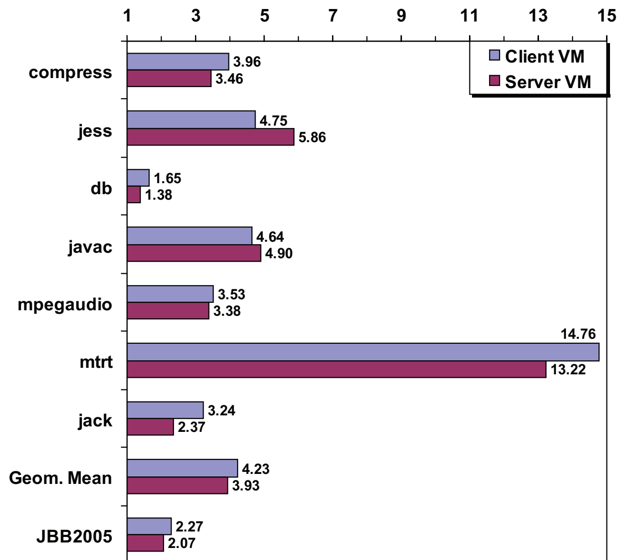

Figure 9: Profiling overhead (slowdown factor) on Sun JDK 1.6.0, 'client' and 'server' modes.

(no networking) and we removed background processes as much as possible in order to obtain reproducible results. For each setting and each benchmark, we took the median of 10 runs. For the SPEC JVM98 suite, we also computed the geometric mean of the 7 benchmarks. Here we present the measurements made with Sun JDK 1.6.0_02-ea-b02 in its 'client' and 'server' modes.

Figure 9 shows the profiling overhead for the two settings. For the SPEC JVM98 benchmarks (resp. the SPEC JBB2005 benchmark), the overhead is computed as a factor of $\frac{\text{execution time with profiling}}{\text{execution time without profiling}}$ $\left(\text{resp. } \frac{\text{operations/second without profiling}}{\text{operations/second with profiling}}\right)$. JP supports user-defined profiling agents to customize profile generation. In all tests, we used a simple profiling agent that avoids processing profiling data during program execution, but employs a JVM shutdown hook to generate the profile upon program termination. On average, the measured overhead due to FERRARI and JP is of factor 3.93–4.23 for the SPEC JVM98 suite, and of factor 2.07–2.27 for SPEC JBB2005.

To compare our approach with a standard profiler based on the JVMPI/JVMTI, we also evaluated the overhead caused by the 'hprof' profiling agent shipped with standard JDKs. We started the profiling agent 'hprof' with the '`-agentlib:hprof=cpu=times`' option, which activates JVMTI-based profiling (available since JDK 1.5.0). The argument '`cpu=times`' ensures that the profiling agent tracks every method invocation, as our instrumentation-code does. For all benchmarks, the overhead caused by 'hprof' is 1–2 orders of magnitude higher than the overhead caused by FERRARI and JP. For 'mtrt' the overhead due to 'hprof' even exceeds factor 3 200 for both the 'client' and the 'server' mode.

In previous work we also promoted an instrumentation-process for sampling profiling, called Komorium [2, 3, 5]. Komorium is fully compatible with FERRARI. With a reasonable sampling rate, FERRARI and Komorium cause an average overhead of about 50%.

7. DISCUSSION

In the following we discuss the strengths and limitations of our approach and outline our ongoing research on dynamic bytecode instrumentation.

As main contribution, our instrumentation framework reconciles full coverage of all bytecodes executed by an application, dynamic instrumentation at runtime, and user-defined instrumentation-processes written in pure Java (and using any Java-based bytecode engineering library). In contrast, prevailing approaches to dynamic bytecode instrumentation either require native code (e.g., JVMTI-based instrumentation [15]) or impose severe restrictions on the instrumentation of certain JDK core classes.

An interesting aspect of our approach is that dynamic instrumentation happens within the same JVM process that runs the program under instrumentation. In order to avoid perturbations of statistics collected by instrumentation-code, our framework offers a mechanism which ensures that dynamic instrumentation (as well as execution of instrumentation-code) is not accounted for. Nonetheless, user-defined instrumentation-processes may leverage the full JDK. A drawback of our approach is that it can perturbate measurements of low-level resource consumption, such as CPU time.

An alternative to our approach would be to execute the instrumentation-process in a separate JVM, as it is done by the NetBeans Profiler [11]. Such a solution ensures that most of the CPU time spent on dynamic instrumentation is consumed by another process. However, using a separate process for dynamic instrumentation suffers from three drawbacks: (1) Overall resource usage (in particular memory consumption) is increased, because the second JVM process has to load and compile at least several hundred JDK classes. (2) In order to instrument also the core classes of the JDK, a JVMTI agent needs to communicate with a separate JVM process using Inter-Process-Communication (IPC) mechanisms of the underlying operating system, which are platform specific. Hence, native code is required and portability is compromised. (3) IPC involves context switches and causes overhead (e.g., cache flush upon context switch). Consequently, significant perturbations are possible.

Another alternative is to do without a separate JVM and directly instrument all classes by a JVMTI agent written in native code. Apart from sacrificing portability, the development of such an instrumentation agent is cumbersome, because to the best of our knowledge, there are no general-purpose bytecode engineering libraries written in native code. Available libraries (e.g., BCEL [9], ASM [12], Javassist [7], JOIE [8], etc.) are all implemented in Java. Actually, an important reason for resorting to a separate JVM for dynamic instrumentation is that implementing an instrumentation-process in native code is much more difficult and error-prone than doing the same in Java with the support of an existing, well-tested bytecode engineering library.

The obvious limitation of our approach is that bytecode instrumentation does not cover native code execution. We can mitigate this problem by instrumenting also native methods using a new feature of the JVMTI in JDK 1.6 called native method prefixing [15, 6]. However, even though we

can instrument invocations of native methods, we still cannot keep track of the native code executed by these methods.

Another drawback of our approach is code bloat. FERRARI employs an instrumentation scheme that keeps a copy of the original method body together with the instrumented version. For the core classes that are statically instrumented, this code duplication cannot be avoided, since during bootstrapping and during dynamic instrumentation the original method bodies have to be executed. For those classes that are dynamically instrumented, the original method bodies will not be executed unless instrumentation-code wants to call these methods with the `execInstrCode` flag disabled (see Section 3.6). In general, FERRARI does not know which methods instrumentation-code is going to call. E.g., in the case of our instrumentation-process JP for exact profiling, instrumentation-code may periodically invoke a custom profiling agent to process profiling data at runtime. There are no restrictions concerning which methods a user-defined profiling agent may call. Nonetheless, we are investigating ways of analyzing custom instrumentation-processes in order to avoid code bloat in dynamically instrumented classes.

As dynamic instrumentation adds to the program execution time, it is important to optimize the instrumentation-process. As a first step, we introduced an alternative API specialized for BCEL-based instrumentation-processes in order to avoid certain inefficiencies in handling bytecode. Our ongoing efforts aim at further reducing the overhead due to dynamic instrumentation.

Applications of FERRARI are not limited to profiling and monitoring. For instance, instrumentation-processes can be defined in order to generate program execution traces, which are needed for various purposes, such as code analysis, testing, reverse engineering, logging, failure diagnosis, etc. FERRARI promises to generate execution traces with much less overhead than prevailing techniques based on the JVMPI [14] or the JVMTI [15].

8. CONCLUSIONS

In this paper we presented a new framework for dynamic bytecode instrumentation in Java. Its particular strength is the support for instrumenting *all* classes, including the core classes of the JDK. This enables bytecode instrumentation to cover the execution of all bytecode in the system. Our framework deals with issues of bootstrapping an instrumented JDK and of avoiding measurement perturbations by runtime instrumentation. It offers a simple API to define custom instrumentation-processes. As shown in this paper, our framework can be used in the area of profiling in order to instrument the whole JDK, resulting in complete, calling-context-sensitive profiles.

9. 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] 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.

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

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

[5] W. Binder and J. Hulaas. Flexible and efficient measurement of dynamic bytecode metrics. In *Fifth International Conference on Generative Programming and Component Engineering (GPCE-2006)*, Portland, Oregon, USA, Oct. 2006.

[6] W. Binder, J. Hulaas, and P. Moret. A quantitative evaluation of the contribution of native code to Java workloads. In *2006 IEEE International Symposium on Workload Characterization (IISWC-2006)*, San Jose, CA, USA, Oct. 2006.

[7] 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.

[8] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

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

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

[11] NetBeans. The NetBeans Profiler Project. Web pages at `http://profiler.netbeans.org/`.

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

[13] Sun Microsystems, Inc. Java Native Interface (JNI). Web pages at `http://java.sun.com/j2se/1.5.0/docs/guide/jni/`.

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

[15] Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. Web pages at `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html`.

[16] 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.

[17] The Standard Performance Evaluation Corporation. SPEC JBB2005 (Java Business Benchmark). Web pages at `http://www.spec.org/osg/jbb2005/`.

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