Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation

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

Jarle Hulaas, Philippe Moret
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH–1015 Lausanne, Switzerland
{jarle.hulaas, philippe.moret}@epfl.ch

Abstract

Java bytecode instrumentation is a widely used technique, especially for profiling purposes. In order to ensure the instrumentation of all classes in the system, including dynamically generated or downloaded code, instrumentation has to be performed at runtime. The standard JDK offers some mechanisms for dynamic instrumentation, which however either require the use of native code or impose severe restrictions on the instrumentation of certain core classes of the JDK. These limitations prevent several instrumentation techniques that are important for efficient, calling context-sensitive profiling. In this paper we present a generic bytecode instrumentation framework that goes beyond these restrictions and enables the customized, dynamic instrumentation of all classes in pure Java. Our framework addresses important issues, such as bootstrapping an instrumented JDK, as well as avoiding measurement perturbations due to dynamic instrumentation or execution of instrumentation code. We validated and evaluated our framework using an instrumentation for exact profiling which generates complete calling context trees of various platform-independent dynamic metrics.

Keywords: *Java, JVM, dynamic bytecode instrumentation, program transformations, dynamic metrics, profiling, aspect-oriented programming*

1. Introduction

Bytecode instrumentation is a valuable technique for transparently enhancing virtual execution environments, for purposes such as monitoring or profiling [36]. In previous work, we applied bytecode instrumentation in order to isolate untrusted software components executing within the same Java Virtual Machine (JVM) [6, 7], to monitor and control resource consumption [11, 14, 25], and to generate calling context-sensitive profiles for performance analysis [8, 10].

In reference [12], we introduced a generic bytecode instrumentation framework called BMW that eases the development of custom instrumentation-processes.[1] BMW is related to aspect-oriented programming [28], since it supports the specification of certain low-level pointcuts (e.g., the beginning of basic blocks of code), where user-defined instrumentation-code templates are inserted, after specialization of the template code according to the individual join points. This specialization, which relies on partial evaluation, can take statically pre-computed bytecode metrics into account. The BMW framework has been designed so as to keep the instrumentation overhead low, and provides specific support for efficiently maintaining thread-local state and for calling context reification, which is particularly useful in profiling use cases.

While we successfully applied the BMW framework in the context of profiling and resource management, the framework suffered from two serious limitations:

1. Core classes of the JDK were not instrumented, resulting in incomplete profiles and resource consumption information.
2. The framework supported only static instrumentation, where the whole application (including libraries) had to be instrumented prior to execution. Hence, dynamically generated or downloaded code was not instrumented.

In this paper we present a generic framework for *dynamic* bytecode instrumentation in *pure Java* called FER-

[1]In order to avoid any confusion between the code that performs an instrumentation and the extra code inserted by an instrumentation, we refer to the former as *instrumentation-process* and to the latter as *instrumentation-code*. *Instrumented-code* denotes the code resulting from an instrumentation-process, i.e., including the original code as well as the inserted instrumentation-code.

RARI (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 be used in conjunction with BMW, although FERRARI can take an arbitrary user-defined instrumentation-process (conforming to the simple FERRARI API, which will be explained later in this paper) and 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 mechanism 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, the number of allocated objects of certain types, etc. 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 [33] or the more recent JVMTI [34].

This paper is structured as follows: Section 2 gives background information on bytecode instrumentation and on existing tools. In Section 3 we describe our generic instrumentation framework, while Section 4 illustrates the API for custom instrumentation-processes. Section 5 presents our concrete use case, a tool for exact profiling that was adapted to exploit the new instrumentation framework. Section 6 discusses the strengths and limitations of our approach. Finally, Section 7 presents related work, and Section 8 concludes this paper.

2. Background

In the following we give an overview of available bytecode instrumentation tools, discuss the pros and cons of static versus dynamic instrumentation, and summarize the instrumentation mechanisms offered by standard JVMs.

2.1. Bytecode Instrumentation Tools

Altering Java semantics via bytecode transformations is a well-known technique [36] 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 [18] represents method bodies as graph structures. Individual bytecode instructions are mapped to Java objects. Both our instrumentation frameworks BMW and FERRARI are based on BCEL.

Javassist [15, 16], which is used by JBoss [27], enables structural reflection and provides convenient source-level abstractions.

Soot [39] is a framework for analyzing and transforming JVM bytecode that offers four intermediate code representations. For instance, *Jimple* is a typed, stack-less, three-address code intermediate represention. Soot is often used for bytecode optimization.

ASM [31], JOIE [17], JikesBT [26], and Serp [5] are further examples of bytecode manipulation libraries implemented in Java.

2.2. Static versus Dynamic Instrumentation

Static bytecode instrumentation inserts all instrumentation-code before the program under instrumentation starts execution. The advantage of this approach is that it causes less runtime overhead, as all classes are instrumented before the program is executed. 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.

2.3. Instrumentation Support in Standard Java Environments

The JVMTI [34] 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 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 [8, 10, 12].

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 aforementioned severe restrictions imposed on class redefinition.

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 [18] 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 [31] or SERP [5]. 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 itself is not instrumented.

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

[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 1. The BootstrapLock class.

instrumentation-process (with the exception of classes in C or in *IP*). For each JVM version and each custom instrumentation-process, the core classes have to be determined (by the probing agent) and instrumented statically.

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 an 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 1). 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 [22], 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 [23].[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 [23].

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 5 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 insert also 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, in particular BMW-based instrumentation-processes [12], indeed insert instance fields in certain classes (e.g., in `java.lang.Thread`), but leave these fields uninitialized. Instrumentation-code is responsible of lazily initializing these fields on demand. Because added instance fields

[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.

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 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 (e.g., in BMW-based instrumentation-processes [12]). 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 such that a new thread 'inherits' the flag value from the creating thread.

Because the constructor of `java.lang.Object` may be frequently invoked by instrumentation-code (e.g., in order to create profiling data structures), FERRARI offers a second mechanism to prevent the 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)`.

[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.

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 5 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 2 and Figure 3 illustrate FERRARI's instrumentation scheme.[5] Figure 2 applies to bootstrapping classes, whereas Figure 3 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.

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.

[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 2. 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 3. Instrumentation scheme for non-bootstrapping classes.

4. API for Custom Instrumentation-Processes

FERRARI uses the `Instrumentation` interface (see Figure 4) in order to invoke a user-defined instrumentation-process. Method `instrument(String, byte[])` takes the fully qualified classname and the class bytecodes as arguments and 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 bytecodes of the added classes. The extra classes are added to the package of the instrumented class.

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

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

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

Figure 4. FERRARI API. The custom instrumentation-process has to implement the Instrumentation interface, while FERRARI provides default implementations of the interfaces Instrumented and MethodDesc.

5. 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 [23] is to use the JVMTI [34] or JVMPI [33] 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 a high 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 obtain detailed execution statistics. JP was first presented in [8], 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` 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).

In the following we evaluate the overhead caused by FERRARI with our JP instrumentation-process for exact profiling in two settings: In the first setting, JP generates a Calling Context Tree (CCT) [2] with the number of method invocations and the number of executed bytecodes for each calling context [8]. In the second setting, each calling context is augmented with additional memory alloction metrics; for details see reference [9].

To evaluate the overhead caused by our profiling scheme, we used the SPEC JVM98 benchmark suite [38] (problem size 100), which consists of 7 benchmarks, as well as the SPEC JBB2005 benchmark [37] (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 (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 the Sun JDK 1.6.0 platform in its 'client' mode.

Figure 5 shows the profiling overhead for the two settings. 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 69–1 386% for the SPEC JVM98 suite, and about 120% for SPEC JBB2005. For all measured benchmarks, the additional overhead for collecting memory allocation metrics is relatively low when compared with the overhead of CCT generation.

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 320 000%.

In previous work we also promoted an instrumentation-process for sampling profiling, called Komorium [8, 10, 12]. Komorium is fully compatible with FERRARI. With a reasonable sampling rate, FERRARI and Komorium cause an average overhead of about 50%.

6. Discussion

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

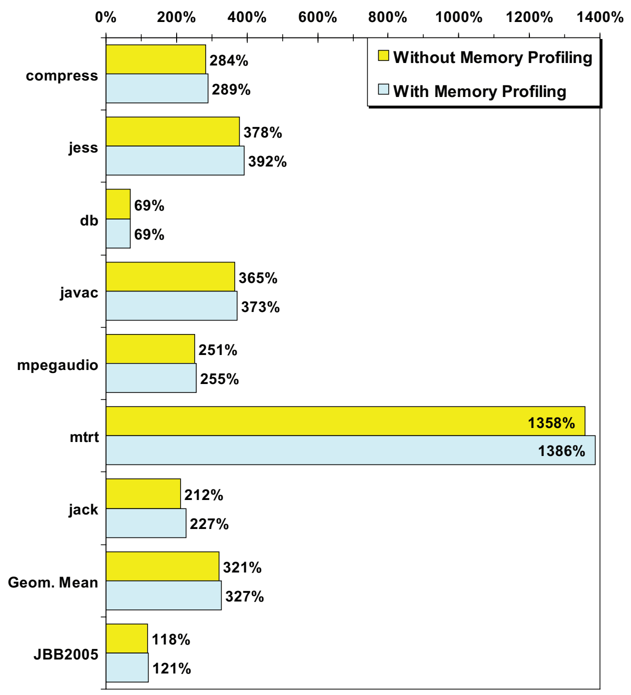

Figure 5. Profiling overhead for SPEC JVM98 and SPEC JBB2005 on Sun JDK 1.6.0, 'client' mode.

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 [34]) 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 [30]. 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 several 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 [18], ASM [31], Javassist [15, 16], Soot [39], JOIE [17], JikesBT [26], Serp [5], 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 [13, 34]. 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. To this end, we are providing a second API, where classes are passed as parsed objects (and not as raw bytecodes) between our framework and the user-defined instrumentation-process. Hence, this approach avoids parsing and dumping the same class twice, but requires the instrumentation-process to employ the same bytecode engineering library our framework uses (i.e., BCEL [18]).

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 [33] or the JVMTI [34].

7. Related Work

While FERRARI is a general-purpose framework for dynamic bytecode instrumentation, it is particularly well suited for implementing profilers. Hence, in the following we discuss related work regarding dynamic metrics and profiling.

In [20] the authors present a variety of dynamic metrics, including bytecode metrics, for selected Java programs, such as the SPEC JVM 98 benchmarks [38]. They introduce a tool called *J [21] for the metrics computation. *J relies on the JVMPI [29, 33], 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, our framework enables instrumentations that incur moderate overhead to 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.

[6]Overheads caused by profiling agents based on the JVMPI [33] or the more recent JVMTI [34] may easily exceed factor 4 000, because certain profiling events prevent runtime optimizations, such as just-in-time compilation [8].

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 [32] 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.

The NetBeans Profiler [30] integrates Sun's JFluid profiling technology [19] into the NetBeans IDE. JFluid exploits dynamic bytecode instrumentation and code hotswapping in order to turn profiling on and off dynamically, for the whole application or just a subset of it. However, this tool needs a customized JVM and is only available for a limited set of environments.

Whereas we perform exact profiling, other approaches use sampling in order to minimize the overhead, but at the price of a possible loss of precision, as e.g. in reference [40]. An interesting technique using counter-based code instrumentation in order to produce profiling samples is described in reference [3]. We also implemented a sampling profiler that approximates the number of executed bytecodes in each calling context [10]. While such an approach reduces overhead, the profiles lack method invocation counters and, depending on the profiled application, may suffer from limited accuracy.

Hardware performance counters that record events, such as instructions executed, cycles executed, pipeline stalls, cache misses, etc. are often exploited for profiling. In [2] hardware performance metrics are associated with execution paths. The Jikes RVM [1], an open source research virtual machine that offers a flexible testbed for prototyping virtual machine technology, has been enhanced to generate traces of hardware performance monitor values for each thread in the system [35]. In [24] 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. In contrast, our instrumentation-process for exact profiling is a software development tool that helps in program analysis. It does not rely on any platform-specific features in order to offer a completely portable profiling system that allows developers to profile applications in their preferred environment, generating reproducible profiles.

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.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, B. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, N. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] 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.

[3] 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.

[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] BEA. Serp. Web pages at `http://serp.sourceforge.net/`.

[6] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, pages 35–42, San Diego, CA, USA, Jan. 2001.

[7] W. Binder. J-SEAL2 – A secure high-performance mobile agent system. *Electronic Commerce Research*, 1(1/2):131–148, 2001.

[8] 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.

[9] W. Binder. Portable profiling of memory allocation in Java. In *Net.ObjectDays 2005 (NODe 2005)*, volume P-69 of *Lecture Notes in Informatics*, pages 110–128, Erfurt, Germany, Sept. 2005.

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

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

[12] 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.

[13] 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.

[14] 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).

[15] 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.

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

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

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

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

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

[21] 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.

[22] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

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

[24] 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.

[25] 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.

[26] IBM. Jikes Bytecode Toolkit. Web pages at `http://www.alphaworks.ibm.com/tech/jikesbt`.

[27] JBoss. Open source middleware software. Web pages at `http://www.jboss.com/`.

[28] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[29] 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.

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

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

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

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

[34] 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`.

[35] 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.

[36] 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.

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

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

[39] 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.

[40] J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, June 2000.