Flexible Calling Context Reification for Aspect-Oriented Programming

Alex Villazón
Faculty of Informatics
University of Lugano
Switzerland
alex.villazon@lu.unisi.ch

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

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

ABSTRACT

Aspect-oriented programming (AOP) eases the development of profilers, debuggers, and reverse engineering tools. Such tools frequently rely on calling context information. However, current AOP technology, such as AspectJ, does not offer dedicated support for accessing complete calling context within aspects. In this paper, we introduce a novel approach to calling context reification that reconciles flexibility, efficiency, accuracy, and portability. It relies on a generic bytecode instrumentation framework ensuring complete bytecode coverage, including the standard Java class library. We compose our program transformations for calling context reification with the AspectJ weaver, providing the aspect developer an efficient mechanism to manipulate a customizable representation of the complete calling context. To highlight the benefits of our approach, we present ReCrash as an aspect using a stack-based calling context representation; ReCrash is an existing tool that generates unit tests to reproduce program failures. In comparison with the original ReCrash tool, our aspect resolves several limitations, is extensible, covers also the standard Java class library, and causes less overhead.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms

Languages, Measurement, Performance

Keywords

Calling context reification, aspect weaving, bytecode instrumentation, composition of program transformations, debugging, Java Virtual Machine

1. INTRODUCTION

Techniques for profiling, debugging, and reverse engineering often benefit from detailed calling context information. Regarding profiling, the *calling context tree* [3] is a popular profiling data-structure that helps locate performance bottlenecks in programs. As an example in the debugging area, calling context information has been used to reproduce the crashing conditions of faulty applications by storing copies of method arguments on a *shadow stack* [4]. Concerning reverse engineering, it is possible to automatically reverse-engineer network protocols by analyzing the handling of different protocol fields in a message, which typically involves different calling contexts [21].

Aspect-oriented programming (AOP) [19] is a promising approach for rapid prototyping of profilers, debuggers, and reverse engineering tools. In [22], several profiling tools are presented as aspects in a concise manner. Unfortunately, current AOP languages, such as AspectJ [18], do not offer dedicated support for efficiently accessing detailed calling context information. While it is possible to specify aspects to gather calling context information, such aspects typically cause high runtime overhead, limiting the practicability of aspect-based techniques for profiling, debugging, or reverse engineering.

In Java, the only standard API to access calling context information is the `Throwable` class. Using `new Throwable().getStackTrace()`, a thread can obtain a trace of its call stack. However, the overhead of allocating a `Throwable` instance and filling in the stack trace can be excessive if it is done frequently, such as upon each method invocation. AOP frameworks, such as JAsCo [27] and JBoss AOP [17], use this technique to implement the `cflow` pointcut, resulting in high overhead [9]. AspectJ provides limited access to dynamic and static calling context information (e.g., `thisJoinPoint`, `thisJoinPointStaticPart`, `thisEnclosingJoinPointStaticPart`). However, there is no efficient, built-in construct to access complete calling context.

In this paper we present a novel program transformation for *customizable*, *efficient*, *accurate*, and *portable calling context reification*. Our approach is *customizable*, because it supports user-defined calling context representations, such as the shadow stack [4]. For *efficiency* reasons, the reified calling context is passed as extra method arguments from the caller to the callee, while ensuring compatibility with native code and with stack introspection. Passing context information as extra arguments is a common technique employed by AspectJ compilers, e.g., for implementing access to the special variable `thisJoinPoint` [15]; the compiler generates code to pass the current `JoinPoint` instance to advices. We generalize the passing of extra arguments to every instrumented method so as to reduce the overhead of complete calling context reification. *Accuracy* of the reified calling context is ensured by using an instrumentation technique that guarantees complete bytecode coverage. That is, every method with a bytecode representa-

AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA.

tion can be instrumented, including methods in the standard Java class library and in dynamically downloaded or generated classes. In order to ensure *portability*, our approach has been implemented in pure Java and has been successfully tested with standard Java Virtual Machines (JVMs) on different platforms.

Our approach integrates particularly well with AspectJ, since it does not require any modification or extension of the AspectJ language and of existing aspect weaving tools. We use a *composition* of FERRARI-based program transformations. FERRARI[1] (Framework for Exhaustive Rewriting and Reification with Advanced Runtime Instrumentation) [7] is a general-purpose bytecode instrumentation framework that generates the necessary program logic to enable *user-defined instrumentation modules (UDIs)* to cover all bytecodes that execute in the JVM. Moreover, we leverage the AJ-UDI [28], which integrates the AspectJ weaver with FERRARI such that aspects can be woven also into the standard Java class library. This feature, which is essential for aspects used in profiling or debugging that require full bytecode coverage, is not supported by current aspect weaving tools, such as AspectJ [18] or *abc* [6].

In this paper we present two new FERRARI UDIs, the CC-UDI and the AJ-CC-UDI. The CC-UDI implements our new approach to calling context reification. The AJ-CC-UDI is a composition of the AJ-UDI and the CC-UDI. In conjunction with FERRARI, the AJ-CC-UDI enhances AspectJ with calling context awareness and full bytecode coverage.

We illustrate the benefits of our approach by representing and extending the functionality of the ReCrash tool [4] with an aspect using a shadow stack. ReCrash is an existing tool (not using AOP) that generates unit tests to reproduce program failures. Thanks to FERRARI and the AJ-CC-UDI, we remove several limitations of ReCrash, significantly improve its coverage, and at the same time reduce its overhead.

The *scientific contributions* of this paper are threefold:

1. We introduce generic program transformations for calling context reification that reconcile customizability, efficiency, accuracy, and portability.

2. We compose these transformations, implemented by the CC-UDI, with the AJ-UDI, enabling aspects with complete bytecode coverage to efficiently manipulate calling context information.

3. We present a case study in the debugging domain, highlighting the benefits of our approach. Moreover, we assess the overhead reductions achieved by using the CC-UDI.

This paper is structured as follows. Section 2 recapitulates our prior work on which this paper builds. Section 3 discusses the limitations of calling context reification with standard AOP constructs and introduces the principles of our efficient calling context reification techniques. Section 4 explores the hurdles that need to be addressed for compatibility with native code and stack introspection. Section 5 details the general instrumentation scheme for calling context reification, implemented by the CC-UDI. Section 6 explains the composition of aspect weaving and calling context reification, provided by the AJ-CC-UDI. Section 7 presents our ReCrash case study. Section 8 assesses the runtime overhead due to calling context reification. Section 9 discusses related work. Finally, Section 10 concludes this paper.

2. PRIOR WORK

In this section we briefly summarize our prior work on which this paper depends; for details, we refer to [7, 28].

[1] `http://www.inf.unisi.ch/projects/ferrari/`

FERRARI [7] is a generic bytecode instrumentation framework that guarantees the instrumentation of every code executing in a virtual machine that has a corresponding bytecode representation. FERRARI consists of a static instrumentation tool and a load-time instrumentation agent, which is based on the `java.lang.instrument` package introduced in JDK 1.5. The former tool is used for statically instrumenting the standard Java class library (including also the vendor-specific Java classes of the runtime system), whereas the latter agent dynamically instruments all application classes. Both of them invoke a user-defined instrumentation module (UDI) through an interface, following the Strategy pattern. The UDI may change method bodies, add new methods (with minor restrictions), and add fields (with some restrictions). To this end, FERRARI passes the original class bytes to the UDI and receives back the UDI-instrumented class bytes. FERRARI's general purpose API [7] allows the seamless integration of existing bytecode transformation tools through UDIs.

FERRARI ensures that UDI-inserted code is not executed before the JVM has completed bootstrapping and provides support for temporarily bypassing the execution of inserted code for each thread, e.g., during load-time instrumentation or when UDI-inserted code invokes instrumented methods, such as methods of the Java class library. FERRARI keeps a copy of the original code of every instrumented method and inserts *bypasses* (conditionals) that allow reverting to the original code.

The AJ-UDI [28] enables AspectJ aspects to be woven into both application and standard Java classes. The AJ-UDI relies on standard, unmodified AspectJ weaving tools. In [28], the AJ-UDI is validated with existing profiling aspects [22] and with an aspect for memory leak detection.

Regarding classes of the standard Java class library, the AJ-UDI imposes some restrictions concerning the possible program transformations. For example, non-singleton aspect instances using `per*` clauses (e.g., per-object or per-control flow aspect association) are currently not supported, because such mechanisms may modify the class hierarchy of the woven classes (e.g., to implement a compiler-generated interface). Changing the hierarchy of JDK classes may break the bootstrapping, therefore the AJ-UDI does not support such constructs. [28] mentions further limitations of the AJ-UDI regarding the inter-type declaration mechanism that allows explicit structural transformations of woven classes. While we have removed many of these restrictions from the AJ-UDI in the meantime, some JVMs may still limit the use of static cross-cutting in JDK classes.[2] Other transformations (method body modifications, insertion of advice invocation, additional static fields, access to reflective information in advices, etc.) are supported for JDK weaving in general.

3. CALLING CONTEXT REIFICATION

AspectJ's reflective API provides information about the current calling context through `thisJoinPoint`, but does not directly provide aspects with the complete calling context. One way to obtain full calling context information is to transform each method so as to maintain a representation of the calling context. We call this approach *calling context reification*.

In this section we discuss two different techniques for calling context reification with aspects. As calling context representation, we focus on the *shadow stack*, which is commonly used in the

[2] For instance, Sun's JVMs make assumptions on the size of the class `Object` that prevent the insertion of instance fields in that class. In contrast, we successfully used inter-type declarations to add instance fields to `Object` in IBM's J9 JVM.

```
public aspect CCAspectSlow {
   public static final class ShadowStack {
      private static final int MAX_STACK_SIZE = 10000;
      public int sp = 0;
      public final Object[] stack = new Object[MAX_STACK_SIZE];
   }

   private static final ThreadLocal<ShadowStack> TL =
      new ThreadLocal<ShadowStack>() {
         protected ShadowStack initialValue() {
             return new ShadowStack();
         }
   };

   pointcut allExecs() :
      execution(* *(..)) && !within(CCAspectSlow);

   before() : allExecs() {
      ShadowStack ss = TL.get();
      ss.stack[ss.sp] = thisJoinPoint;  // push context
      ss.sp = ss.sp + 1;
   }

   after() returning() : allExecs() {
      ShadowStack ss = TL.get();
      ss.stack[ss.sp] = null;           // pop context
      ss.sp = ss.sp - 1;
   }

   after() throwing() : allExecs() {
      ShadowStack ss = TL.get();
      ss.stack[ss.sp] = null;           // pop context
      ss.sp = ss.sp - 1;
   }
}
```

Figure 1: Naive calling context reification.

area of debugging [4]. Each thread maintains a separate, thread-confined shadow stack reflecting the invoked methods on the call stack in the virtual machine. First, we present a naive approach to calling context reification that relies only on standard AspectJ constructs. However, the naive approach suffers from high overhead. Therefore, we introduce our optimized approach, which requires special program transformations in addition to aspect weaving.

3.1 Naive Reification Using a Thread-local Variable

AOP allows us to concisely specify program transformations for calling context reification. Figure 1 illustrates an aspect `CCAspectSlow` that maintains a shadow stack for each thread. The shadow stack is an instance of type `ShadowStack` that includes an object array `stack` and an integer stack pointer `sp`; the next free entry on the shadow stack is `stack[sp]`. We assume that `MAX_STACK_SIZE` is large enough such that the shadow stack never overflows. The thread-local variable `TL` provides the `ShadowStack` instance of the current thread.

Upon method entry and completion (normal completion as well as abnormal completion by throwing an exception), the current thread's shadow stack is updated accordingly. On method entry, a representation of the invoked method is pushed on the shadow stack; upon method completion, it is popped off the shadow stack. We use `JoinPoint` instances to represent method invocations on the shadow stack (provided by the `thisJoinPoint` construct).

The aspect in Figure 1 has three advices. The first advice handles method entry. The other two advices handle normal respectively abnormal method completion. Here we do not consider the execution of constructors in order to simplify the discussion; in our case study in Section 7 we will also deal with constructors.

The `CCAspectSlow` shown in Figure 1 has two major drawbacks:

```
public aspect CCAspectFast {
   pointcut allExecs() :
      execution(* *(..)) && !within(CCAspectFast);

   before() : allExecs() {
      SSAccess.thisStack()[SSAccess.thisSP()] = thisJoinPoint;
   }

   after() returning() : allExecs() {
      SSAccess.thisStack()[SSAccess.thisSP()] = null;
   }

   after() throwing() : allExecs() {
      SSAccess.thisStack()[SSAccess.thisSP()] = null;
   }
}

public final class SSAccess { // marker methods
   public static Object[] thisStack() {
      throw new UnsupportedOperationException();
   }

   public static int thisSP() {
      throw new UnsupportedOperationException();
   }
}
```

Figure 2: Efficient calling context reification.

1. The shadow stack excludes invocations of native methods, because the aspect can be woven only into Java methods.

2. The aspect causes high runtime overhead, as we will show in Section 8, because the thread-local variable `TL` is accessed in each advice. Moreover, the stack pointer `sp`, which is kept in an object field on the heap, is updated upon method entry as well as upon method completion, contributing to the high overhead.

The first limitation can be solved with *native method prefixing*, a feature of the standard JVM Tool Interface (JVMTI) [26] introducted in JDK 1.6. Native methods are renamed by prepending a well-chosen prefix that is announced to the JVM (the prefix should not occur in any method name). When linking native code libraries, the JVM is able to match method names declared with a prefix with unchanged method names in native code libraries. For each renamed native method, a Java method with the original name and signature is added, which invokes the corresponding renamed native method; we call these inserted Java methods *replacement methods*. Since replacement methods are instrumented by the aspect weaver as any other Java methods, the generated calling context representation will include the invocations of native methods. Our approach to calling context reification presented in the next sections relies on native method prefixing.

The second issue, high runtime overhead, can be addressed by passing the caller's context (i.e., the object array `stack` and the stack pointer `sp`) to the callee as additional method arguments. Instead of maintaining the calling context as a `ShadowStack` instance on the heap that is accessed through a thread-local variable, each method invocation keeps a reference to the object array `stack` and its `sp` value in local variables on the call stack. Since it is not possible to express such an instrumentation with current AOP languages, we implemented it as a special FERRARI UDI for calling context reification (CC-UDI) and composed that UDI with the AspectJ weaver.

3.2 Efficient Reification with Extra Method Arguments

Figure 2 shows an aspect for efficient calling context reification. It relies on a dedicated instrumentation of the woven code (provided by the CC-UDI) so as to pass the calling context as extra method arguments. Moreover, it requires the compiled aspect itself

Compiled CCAspectSlow (naive reification):

```
public class CCAspectSlow {
   public static CCAspectSlow aspectOf() {...}

   public void before(JoinPoint tjp) {
      ShadowStack ss = TL.get();
      ss.stack[ss.sp] = tjp;
      ss.sp = ss.sp + 1;
   }

   public void afterReturning() {
      ShadowStack ss = TL.get();
      ss.stack[ss.sp] = null;
      ss.sp = ss.sp - 1;
   }

   public void afterThrowing() {
      ShadowStack ss = TL.get();
      ss.stack[ss.sp] = null;
      ss.sp = ss.sp - 1;
   }
   ...
}
```

Compiled and transformed CCAspectFast (efficient reification):

```
public class CCAspectFast {
   public static CCAspectFast aspectOf() {...}

   public void before(JoinPoint tjp, Object[] stack, int sp) {
      stack[sp] = tjp;
   }


   public void afterReturning(Object[] stack, int sp)  {
      stack[sp] = null;
   }


   public void afterThrowing(Object[] stack, int sp) {
      stack[sp] = null;
   }


   ...
}
```

Figure 3: Comparison of the compiled CCAspectSlow with the compiled and transformed CCAspectFast (pseudo-code).

Sample code calculating Fibonacci numbers:

```
public class Fibonacci {
   public static int fib(int n) {
      // assert n >= 0;
      return n<=1 ? n : fib(n-1)+fib(n-2);
   }
}
```

Code after weaving CCAspectSlow:

```
public class Fibonacci {
   public static int fib(int n) {

      JoinPoint tjp = ...; // create JoinPoint instance
      CCAspectSlow a = CCAspectSlow.aspectOf();
      try {
         a.before(tjp);
         int res = (n<=1 ? n : fib(n-1)+fib(n-2));

         a.afterReturning();
         return res;
      }
      catch(Throwable t) {
         a.afterThrowing();
         throw t;
      }
   }



}
```

Code after weaving CCAspectFast and after instrumentation:

```
public class Fibonacci {
   public static int fib(int n, Object[] stack, int sp) {
      int calleeSP = sp + 1;
      JoinPoint tjp = ...; // create JoinPoint instance
      CCAspectFast a = CCAspectFast.aspectOf();
      try {
         a.before(tjp, stack, sp);
         int res = (n<=1 ? n : fib(n-1, stack, calleeSP) +
                               fib(n-2, stack, calleeSP));
         a.afterReturning(stack, sp);
         return res;
      }
      catch(Throwable t) {
         a.afterThrowing(stack, sp);
         throw t;
      }
   }

   public static int fib(int n) {
      return fib(n, SSRuntime.findStack(), SSRuntime.findSP());
   }
}
```

Figure 4: Comparison of a woven method using CCAspectSlow with a woven and instrumented method using CCAspectFast (pseudo-code).

to be transformed in order to access the introduced extra arguments (details of this transformation will be presented in Section 6).

`CCAspectFast` gets direct access to the object array `stack` and to the stack pointer `sp` through the static methods `SSAccess.thisStack()` and `SSAccess.thisSP()`. These methods are used as markers for the transformation of the compiled aspect and are never invoked at runtime (this is emphasized in the method bodies throwing `UnsupportedOperationException`). In the compiled advice methods, all invocations of these two static methods are replaced with bytecodes that access the corresponding extra arguments.

Figure 3 compares `CCAspectSlow` after compilation with `CCAspectFast` after compilation and transformation. While `CCAspectSlow` has to access the thread-local variable `TL` upon each advice invocation, `CCAspectFast` directly updates the object array `stack` at the right position. Note that `CCAspectFast` does not update the stack pointer `sp`, since our instrumentation (which is applied to the woven code) will pass the correct `sp` value to each invoked advice method. The pseudo-code in Figure 4 illustrates the application of `CCAspectSlow` and `CCAspectFast` to a sample method computing Fibonacci numbers. The aspect weaver generates infrastructural code to create the context information at the join

point, which is passed to the `before` advice, since it makes use of `thisJoinPoint`. The weaver also inserts invocations to the static `aspectOf()` method (which is generated upon aspect compilation) to access the singleton aspect instance.[3]

For `CCAspectFast`, the CC-UDI (which is applied after aspect weaving) introduces the extra method arguments, computes the stack pointer value `calleeSP` to be passed to callee methods, and generates a method with the original signature to ensure compatibility with native code (and reflection), which is not aware of the extra arguments and hence may invoke the method with the original signature. In that case, the static methods `SSRuntime.findStack()` and `SSRuntime.findSP()` look up the current thread's shadow stack. The implementations of these methods will be given in Section 5.

When compared to `CCAspectSlow`, `CCAspectFast` reduces the overhead of calling context reification by almost factor 2 in standard, state-of-the-art virtual machines. A detailed overhead evaluation will be presented in Section 8.

4. METHOD OVERLOADING

In this section we explore the intricacies of introducing the reified calling context as extra method arguments. First, we consider native code compatibility, which requires method overloading by our instrumentation. Afterwards, we investigate limitations of method overloading.

While in the previous section we concentrated on the shadow stack and represented the calling context as two extra arguments (an object array and an integer), we now consider a generalized calling context represented by an N-tuple $< cc_1, ..., cc_N >$, where cc_i is of type CC_i; CC_i may be a primitive type or a reference type ($1 \leq i \leq N$). Each element cc_i of the tuple is mapped to an extra argument. For the shadow stack, N=2, CC_1=`Object[]`, and CC_2=`int`.

4.1 Compatibility with Native Code

Conceptually, the signatures of all methods, including the inserted replacement methods (due to native method prefixing), are extended with the N extra arguments of types CC_i so as to pass the calling context from the caller to the callee. Hence, a non-native method `f()` gets transformed into method `f(`CC_1`,...,`CC_N`)`, which we call *Extended-Signature-Instrumented (ESI)* method.

For compatibility with native code, it is essential to provide methods with the unmodified signature such that native code can invoke any Java method without passing the extra arguments. We call these introduced methods with the original signature *Native-to-Bytecode (N2B)* wrappers.

As we rely on native method prefixing [26], native methods are invoked only by the inserted replacement methods. Replacement methods do not pass any extra argument upon invocation of the corresponding native methods.

N2B wrappers (as well as prefixed native methods) preserve the original method signature, whereas ESI methods receive N extra arguments. That is, conceptually we are *overloading* all methods, including also method declarations in interfaces.

4.2 Non-Overloadable and Non-Wrappable Methods

Ideally, all methods should be overloaded in the same way. Unfortunately, in current JDKs there are a few methods that cannot be overloaded; we call them *non-overloadable* methods. Non-overloadable methods are not necessarily excluded from instrumentation; normally, the bodies of non-overloadable Java methods can be instrumented.

A few JDK methods cannot be overloaded for three reasons:

1. A bug in Sun's recent JDKs prevents the insertion of more than two non-static, non-private, non-final methods into `java.lang.Object`[4]. We work around this bug by making `Object.toString()` and `Object.equals(Object)` non-overloadable.

2. We consider `Object.finalize()` non-overloadable and also exclude it from instrumentation. Otherwise, the overhead of garbage collection would significantly increase for each object.

3. Some code for stack introspection within the JVM verifies the signatures of certain methods on the call stack. Since ESI methods have extended signatures, their presence on the call stack may break such checks. In Sun's recent JDKs, the method `java.lang.reflect.Method.invoke(...)` is non-overloadable for this reason.

Because non-overloadable methods must be invoked without extra arguments, it is necessary to consider polymorphic call sites and inheritance of non-overloadable methods. FERRARI [7] provides dedicated support for determining whether extra arguments can be passed upon instrumentation of a call site.

The use of stack introspection in the JVM causes further complications. Invocations of N2B wrappers constitute extra frames on the call stack. The presence of such additional stack frames may break JVM-internal code for stack introspection. For instance, in Sun's JDKs there are certain methods that rely on a particular invocation sequence. Examples include methods in `java.lang.Class`, `java.lang.ClassLoader`, `java.lang.Runtime`, and `java.lang.System`. The execution of the affected methods requires inspection of the n topmost stack frames of callers to determine whether an operation shall be permitted; for each affected method, n is a statically known constant. Additional stack frames due to the invocation of N2B wrappers may break such stack introspecting code. Those methods in the Java class library that must not be invoked through wrappers are called *non-wrappable* methods in the following.

Invocations of replacement methods also constitute extra frames on the call stack. However, since the Java class library supports native method prefixing, the code for stack introspection skips the frames corresponding to prefixed native methods. Nonetheless, if a native method is non-wrappable, then the corresponding replacement method is non-wrappable.

We need to make sure that there are no extra stack frames when invoking a non-wrappable method. This can be achieved by using code duplication instead of wrapping for compatibility with native code. That is, instead of introducing an N2B wrapper that calls the corresponding ESI method with the extra arguments, we can duplicate the instrumented method body in the method with the unmodified signature; we call the resulting method an *Original-Signature-Instrumented (OSI)* method.

In Sun's recent JDKs, there are about 50 non-wrappable Java methods where OSI methods must be generated instead of N2B wrappers. If the set of non-wrappable methods is not known, it is possible to use OSI methods instead of N2B wrappers for all methods, causing however significant code bloat.

[3]The weaver inserts code to call `aspectOf()` before each advice invocation, which is not shown in Figure 4 for simplicity.

[4]`http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6583051`

```
void f() {
   ...
   aspectOf();
   ...
   advice();
   ...
   overloadable();
   ...
   nonoverloadable();
   ...
}
```

Figure 5: Example woven method to be instrumented for calling context reification (pseudo-code).

5. INSTRUMENTATION FOR CALLING CONTEXT REIFICATION (CC-UDI)

In this section we present the details of our instrumentation for efficient, accurate, and portable calling context reification, which we implemented as a configurable FERRARI UDI, called CC-UDI. Here we describe the transformation scheme assuming previous aspect weaving. The actual composition of aspect weaving with the CC-UDI instrumentation will be illustrated in the next section.

5.1 General Instrumentation Scheme

As mentioned before, the CC-UDI represents the calling context as an N-tuple $< cc_1, ..., cc_N >$, where cc_i is of type CC_i. N and the types CC_i are configuration parameters of the CC-UDI.

In addition, the CC-UDI has the configuration parameters *findContext*$_i$ and *calleeContext*$_i$, which refer to user-defined static methods that are inlined in the instrumented code. We call these static methods *code snippets*. Code snippet *findContext*$_i$ has no arguments and returns an instance/value of type CC_i. It is inlined in N2B wrappers and in OSI methods in order to obtain cc_i. Code snippet *calleeContext*$_i$ takes and returns an instance/value of type CC_i. It is inlined in ESI and OSI methods so as to compute the i^{th} extra argument to be passed to callees.

Figure 5 shows a woven example method `f()` with invocations to the `aspectOf()` method, to an advice method, to an overloadable method, and to a non-overloadable method. Invocations of native methods within replacement methods are treated in the same way as non-overloadable methods (i.e., extra arguments are not passed).

The pseudo-code in Figure 6 illustrates the CC-UDI instrumentation for N2B, ESI, respectively OSI methods. `aspectOf()` and non-overloadable methods are invoked without extra arguments. Advice methods receive the extra arguments that represent the calling context of the currently executing method, whereas overloadable methods receive the extra arguments representing the callee's context. Depending on the characteristics of method `f()` (overloadable versus non-overloadable, wrappable versus non-wrappable), N2B, ESI, respectively OSI methods are generated as follows:

Method f()	Generated methods
Overloadable and wrappable	N2B and ESI
Overloadable and non-wrappable	OSI and ESI
Non-overloadable	OSI

5.2 Inlining of Code Snippets

The instrumentation performed by the CC-UDI requires the inlining of the code snippets *findContext*$_i$ and *calleeContext*$_i$ according to Figure 6. Here we give an overview of the necessary instrumentation steps.

Since for load-time instrumentation of application classes, the execution of the inlining algorithm contributes to the runtime overhead, the algorithm should be as efficient as possible. Hence, the

N2B:

```
void f() {
   CC_1 cc_1 = findContext_1(); // inlined
   ...
   CC_N cc_N = findContext_N(); // inlined

   f(cc_1, ..., cc_N);
}
```

ESI:

```
void f(CC_1 cc_1, ..., CC_N cc_N) {
   CC_1 calleeCC_1 = calleeContext_1(cc_1); // inlined
   ...
   CC_N calleeCC_N = calleeContext_N(cc_N); // inlined

   ...
   aspectOf();
   ...
   advice(cc_1, ..., cc_N);
   ...
   overloadable(calleeCC_1, ..., calleeCC_N);
   ...
   nonoverloadable();
   ...
}
```

OSI:

```
void f() {
   CC_1 cc_1 = findContext_1(); // inlined
   ...
   CC_N cc_N = findContext_N(); // inlined

   CC_1 calleeCC_1 = calleeContext_1(cc_1); // inlined
   ...
   CC_N calleeCC_N = calleeContext_N(cc_N); // inlined

   ...
   aspectOf();
   ...
   advice(cc_1, ..., cc_N);
   ...
   overloadable(calleeCC_1, ..., calleeCC_N);
   ...
   nonoverloadable();
   ...
}
```

Figure 6: Generated N2B, ESI, respectively OSI methods for method f() in Figure 5.

CC-UDI imposes three restrictions on the code snippets in order to simplify inlining.

1. The JVM local variable corresponding to the argument of code snippet *calleeContext*$_i$ is only read but not written. Conceptually, *calleeContext*$_i$ is a function that does not modify its argument.
2. Each code snippet leaves the operand stack empty on return.
3. In each code snippet, the last instruction is a return bytecode, and there is no second return bytecode in the snippet. Thus, when inlining a code snippet, we can simply delete the return bytecode form the snippet, and the inlined code will leave the return value on the operand stack.

For the code snippets we have been using with the CC-UDI so far, standard Java compilers generate bytecode conforming to these restrictions. Otherwise, the code snippets can be provided as manually crafted bytecode. Furthermore, code snippets may invoke arbitrary methods, which are not concerned by the aforementioned restrictions.

For each extra argument i, the CC-UDI allocates two JVM local variables, $lvar_i^{caller}$ and $lvar_i^{callee}$, in the method under instrumentation. While $lvar_i^{caller}$ holds the instance/value cc_i provided by the caller (in ESI methods) or by the code snippet *findContext*$_i$ (in OSI

methods), $lvar_i^{callee}$ stores the instance/value computed by the code snippet $calleeContext_i$, which will be passed to overloadable callee methods. Note that in ESI methods, $lvar_i^{caller}$ must correspond to the JVM local variable in which the i^{th} extra argument is passed. If that local variable is used in the original method body, the body needs to be updated to use a different local variable instead.

The CC-UDI performs two straightforward optimizations to reduce the number of allocated JVM local variables and to avoid generating unnecessary bytecode.

1. If the method under instrumentation was a leaf method before aspect weaving (i.e., disregarding invocations of `aspectOf()` and of advice methods), then for all i, the local variable $lvar_i^{callee}$ is not allocated and the code snippet $calleeContext_i$ is not inlined.

2. If code snippet $calleeContext_i$ computes the identity function (i.e., it simply returns the passed argument of type CC_i), then $lvar_i^{callee}$ is not allocated and the code snippet $calleeContext_i$ is not inlined. Overloadable callee methods will receive the instance/value stored in $lvar_i^{caller}$ as i^{th} extra argument.

The code snippet $findContext_i$ is inlined as follows: First, rename all local variables used in the code snippet such that they do not interfere with the method under instrumentation. Second, delete the trailing return bytecode in the code snippet. Third, copy the remaining bytecodes of the code snippet and insert a subsequent bytecode to store the value on top of the operand stack in the local variable $lvar_i^{caller}$.

Inlining of the code snippet $calleeContext_i$ proceeds in a similar way. However, local variable zero in the code snippet must be treated specially, since it is supposed to receive the argument cc_i. Thus, in the snippet we rename local variable zero to $lvar_i^{caller}$. The result of the code snippet on the operand stack is stored in $lvar_i^{callee}$.

5.3 Configuration for the Shadow Stack

For the shadow stack presented in Section 3, the reified calling context is represented by two extra arguments, an object array `stack` and an integer stack pointer `sp`. The configuration of the CC-UDI, the code snippets in class `SSCodeSnippets`, and the runtime class `SSRuntime` are given in Figure 7.

Note that the code snippet `calleeStack(Object[])` computes the identity function; hence, only a single JVM local variable will be allocated for the object array `stack`, and the snippet `calleeStack(Object[])` will not be inlined.

The object array `stack` is kept in the thread-local variable `SSRuntime.TL`. Thus, in N2B wrappers and in OSI methods, the `stack` can be retrieved from the thread-local variable. In contrast, the stack pointer `sp` is never stored in a thread-local variable, but is computed by searching the `stack` for the `null` value with the lowest index (binary search). Note that access to the thread-local variable and computation of the stack pointer happen relatively infrequently, because in the common case of an overloadable method, the caller directly passes `stack` and the correct `sp` value as extra arguments.

6. COMPOSITION OF TRANSFORMATIONS

In this section we firstly illustrate the overall code transformations performed by the AJ-CC-UDI that enables woven aspects to access complete calling context information. Thanks to FERRARI [7], complete bytecode coverage is ensured. Secondly, we discuss the necessary transformations of the compiled aspect. Figure 8 gives an overview of the transformation steps.

```
N              = 2;
CC_1           = Object[];
CC_2           = int;
findContext_1  = SSCodeSnippets.findStack();
findContext_2  = SSCodeSnippets.findSP();
calleeContext_1 = SSCodeSnippets.calleeStack();
calleeContext_2 = SSCodeSnippets.calleeSP();

public final class SSCodeSnippets { // inlined by the CC-UDI
   public static Object[] findStack() {
      return SSRuntime.findStack();
   }

   public static int findSP() { return SSRuntime.findSP(); }

   public static Object[] calleeStack(Object[] stack) {
      return stack;
   }

   public static int calleeSP(int sp) { return sp + 1; }
}

public class SSRuntime { // called by inlined code snippets
   private static final int MAX_STACK_SIZE = 10000;

   private static final ThreadLocal<Object[]> TL =
       new ThreadLocal<Object[]>() {
          protected Object[] initialValue() {
             return new Object[MAX_STACK_SIZE];
          }
   };

   public static Object[] findStack() { return TL.get(); }

   public static int findSP() {
      Object[] stack = findStack();
      int min = 0;
      int max = stack.length - 1;
      assert stack[max] == null; // the stack is never full

      // find the smallest index i such that stack[i] == null
      while (min < max) { // binary search
         int mid = (min + max) / 2;
         if (stack[mid] == null) max = mid;
         else min = mid + 1;
      }
      return min;
   }
}
```

Figure 7: Shadow stack: CC-UDI configuration, code snippets, and class SSRuntime.

6.1 FERRARI Instrumentations (AJ-CC-UDI)

The AJ-CC-UDI is a composition of the AJ-UDI [28] and the CC-UDI presented in the previous section.

Figure 9 illustrates the dynamic behavior of the program transformation composition implemented by the AJ-CC-UDI for load-time instrumentation (for the static instrumentation of JDK classes, the AJ-CC-UDI behaves in a similar way). FERRARI's instrumentation agent receives the class to be instrumented through the `java.lang.instrument` API. All the parameters and return values shown in Figure 9 are byte arrays representing a class. The overall instrumentation involves the following four steps:

1. *Native method prefixing:* FERRARI introduces a replacement method for each native method.

2. *Aspect weaving:* The AJ-UDI weaves the aspect into Java methods (including the replacement methods).

3. *Calling context reification:* The CC-UDI overloads methods to pass the reified calling context as extra arguments whenever possible, according to a given configuration.

4. *Bypass generation:* FERRARI generates bypasses for skipping UDI-inserted code during JVM bootstrapping, during load-time instrumentation, and when executing methods of the aspect.

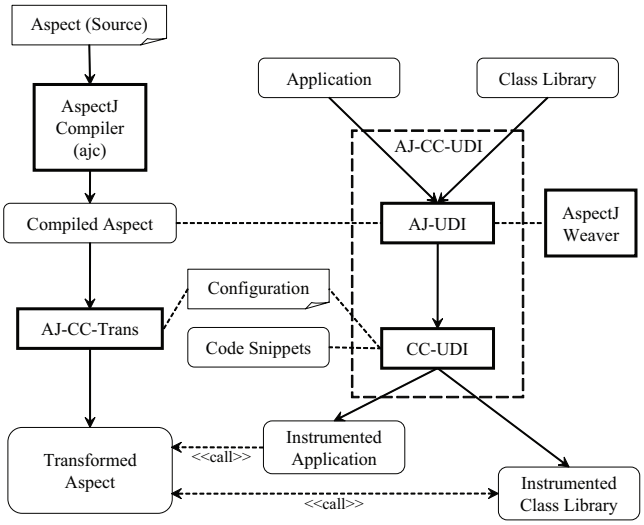

Figure 8: Overview of the transformation steps.

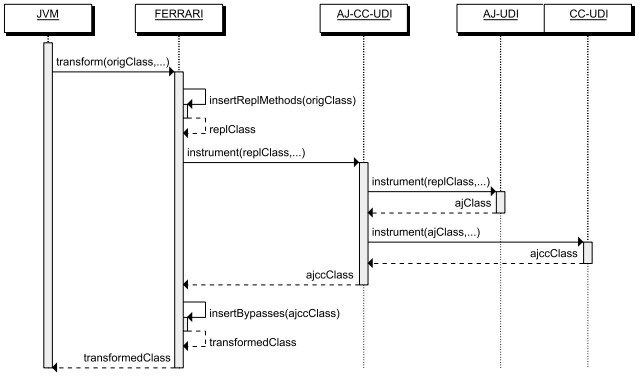

Figure 9: Simplified sequence diagram of the instrumentation composition.

6.2 Aspect Transformations (AJ-CC-Trans)

The compiled aspect (which is not processed by FERRARI) is transformed by the automated tool AJ-CC-Trans in order to handle the reified calling context.

AJ-CC-Trans relies on the CC-UDI configuration parameters N (the number of extra arguments) and CC_i (the types of the extra arguments). In addition, it requires additional configuration parameters $accessContext_i$, which refer to static marker methods to access the i^{th} extra argument within aspects. In the case of the shadow stack, $accessContext_1$=`SSAccess.thisStack()` and $accessContext_2$=`SSAccess.thisSP()`. Figure 2 gives an example how the calling context is accessed within an aspect.

AJ-CC-Trans transforms the compiled aspect in the following three ways:

1. The signatures of the advice methods are extended to receive the reified calling context as extra arguments (the signature of the `aspectOf()` method is not modified). This transformation works, because the CC-UDI, which is applied after aspect weaving, transforms invocations to the advice methods so as to pass the calling context as extra arguments.

2. Invocations of the static marker methods $accessContext_i$ are replaced with corresponding JVM load bytecodes.

3. In the beginning of each advice method, code is inserted to activate the bypasses introduced by FERRARI such that the execution of advices does not create any artifacts in the reified calling context. Hence, advice methods may invoke methods in the Java class library without risking any recursive advice invocation.

7. CASE STUDY: RECRASH

ReCrash [4] is a tool enabling developers to analyze and reproduce the state of an application before crashing. It generates unit tests that help the developer recreate crash conditions. ReCrash instruments the application code to store (copies of) the actual arguments passed to every method on a shadow stack. Upon method entry, an element containing the arguments is pushed on the shadow stack and upon normal method termination, the element is removed from the shadow stack. ReCrash can use different copy strategies; e.g., it can make a deep copy of each argument or store only a reference. Crashes are detected as uncaught exceptions in the `main(String[])` method. The original `main(String[])` method is wrapped with a special exception handler that generates the unit test to reproduce the crash using the elements of the shadow stack.

The current implementation of ReCrash suffers from several limitations:

1. *Functional flaws:*
   - If an exception thrown in a callee method is caught by a caller, the shadow stack is not updated accordingly, resulting in spurious elements on the shadow stack. Upon a subsequent crash, test cases may be generated that are not related to the crash.
   - Crashes within constructors are not handled.
   - Crashes in threads other than the main thread are not handled.
2. *Hard-coded instrumentation:* Extending ReCrash is difficult, since the tool relies on ASM[5], a rather low-level bytecode manipulation library.
3. *Incomplete shadow stack:* ReCrash does not support instrumentation of the standard Java class library. Thus, it is of limited use for JDK developers.
4. *High overhead:* When applied to standard benchmarks (see Section 8), ReCrash introduces high overhead.[6]

Figure 10 presents our `ReCrashAspect` that solves all the aforementioned limitations, thanks to FERRARI and the AJ-CC-UDI. `ReCrashAspect` is similar to `CCAspectFast` in Figure 2. However, there are two major differences: First, the `ReCrashAspect` covers the execution of constructors, which are also represented on the shadow stack. The `preinitialization` pointcut ensures that the shadow stack is updated in the very beginning of a constructor, before invoking another constructor of the same class or of the superclass. Second, exception handling is different, because the shadow stack must not be updated upon abnormal method completion, unless the exception is caught by a caller. Otherwise, in the case of a crash, the shadow stack would be empty and useless for test case generation.

The advice woven in the beginning of exception handlers (`before(Throwable e): handler(*) && args(e)`) nullifies the topmost shadow stack elements corresponding to the callees that completed abnormally. Note that this is only possible because each method on the call stack holds its corresponding `sp` value in a local variable. In the case of `CCAspectSlow` in Figure 1, a correct cleanup of the shadow stack would be impossible, since there is only a single `sp` value stored in the field of an object on the heap,

[5] `http://asm.objectweb.org/`

[6] The authors of ReCrash reported significantly lower overhead [4]. However, instead of standard benchmarks, they used only a few applications, such as "SVNKit checkout", which were likely I/O-bound.

```
public aspect ReCrashAspect {
   pointcut allExecsNews() :
      (execution(* *(..)) || execution(*.new(..)))
      && !within(ReCrashAspect)
      && !within(com.throughtworks..*)    // used by ReCrash's
      && !within(edu.mit.csail.pag.recrash..*); // TraceWriter

   pointcut allExecsPreinits() :
      (execution(* *(..)) || execution(*.new(..))
       || preinitialization(*.new(..)))
      && !within(ReCrashAspect)
      && !within(com.throughtworks..*)    // used by ReCrash's
      && !within(edu.mit.csail.pag.recrash..*); // TraceWriter

   before() : allExecsPreinits() {
      SSAccess.thisStack()[SSAccess.thisSP()] = thisJoinPoint;
   }

   after() returning() : allExecsNews {
      SSAccess.thisStack()[SSAccess.thisSP()] = null;
   }

   before(Throwable e) : handler(*) && args(e) {
      Object[] stack = SSAccess.thisStack();
      int sp = SSAccess.thisSP() + 1;
      while (stack[sp] != null) // cleanup upon caught exception
         stack[sp++] = null;
   }

   after() throwing(Throwable e) :
      execution(public static void *.main(String[])) {
         doReCrash(SSAccess.thisStack(), e);
   }

   private static void doReCrash(Object[] stack, Throwable e) {
      for (int i = 0; stack[i] != null; i++) {
         JoinPoint jp = (JoinPoint)stack[i];
         Object[] oar = jp.getArgs(); // method arguments
         Object[] ar = new Object[oar.length + 1];
         System.arraycopy(oar, 0, ar, 1, oar.length);
         ar[0] = jp.getThis(); // null if static method
         String[] st = new String[ar.length];
         Signature sigjp = jp.getSignature();
         st[0] = sigjp.getDeclaringTypeName();
         for (int j = 1; j < st.length; j++) {
            st[j] = ar[j].getClass().getName();
         }
         boolean[] b = new boolean[st.length]; // no deep copies
         String sig = st[0] + "." + sigjp.getName();
         TraceWriter.youMayCrash(sig, ar, st, b);
      }
      TraceWriter.writeTrace(e); // generate unit test cases
   }
}
```

Figure 10: ReCrashAspect using a shadow stack.

and the advice cannot know the number of callees that completed abnormally (i.e., the number of elements to pop off the shadow stack).[7]

Crashes are detected as uncaught exceptions in the `main(String[])` method. This is captured by the `after() throwing(Thowable e)` advice which adds an exception handler to `main(String[])`. In the case of a crash, the `ReCrashAspect` processes the `JoinPoint` instances on the shadow stack; a `null` value indicates the top of the shadow stack. The details of the method `doReCrash(...)` are unimportant; it uses the `TraceWriter` functionality provided by the original ReCrash tool to generate the unit tests.

[7]An alternative implementation of `CCAspectSlow` using an `around` advice could keep a local copy of the `sp` value upon method entry. The *exception introduction pattern* [20] could be used to propagate exceptions. However, our measurements have shown that such a solution causes even higher overhead than `CCAspectSlow`.

```
after() throwing(Throwable e) : execution(void Thread+.run()) {
   doReCrash(SSAccess.thisStack(), e);
}
```

Figure 11: Extension of ReCrashAspect to handle crashes in threads different from the main thread.

Compared to the original ReCrash tool, our aspect offers the following enhancement:

1. It solves the aforementioned functional flaws. The shadow stack is correctly maintained in the case of an exception caught by a caller. Crashes in constructors are correctly handled, too.

2. The aspect can be easily extended. For instance, crashes in threads other than the main thread can be handled by adding the advice shown in Figure 11.

3. Since the instrumentation performed by the AJ-CC-UDI covers every method in the virtual machine that has a bytecode representation, the shadow stack includes the invocations of methods in the standard Java class library. Hence, our aspect can be a valuable tool also for JDK developers.

4. As we will show in Section 8, our `ReCrashAspect` outperforms the original ReCrash tool on standard benchmarks.

8. EVALUATION

In this section we present our evaluation results. First, we assess the execution time overhead caused by our approach to efficient calling context reification and compare it with the overhead due to naive calling context reification as discussed in Section 3. Second, we evaluate our `ReCrashAspect` and compare it with the original ReCrash tool[8] (version 0.3).

For our evaluation, we use the SPEC JVM98[9] benchmark suite (problem size 100) and compute the geometric mean of the benchmarks' execution times. Our test platform is a Linux Fedora Core 2 computer (Intel Pentium 4, 2,66 GHz, 1024 MB RAM). We present measurements made with the Sun JDK 1.7.0-ea-b25 HotSpot Client and Server VMs. The presented measurements correspond to the median of 15 runs within the same JVM process in order to attenuate the perturbations due to load-time instrumentation (both FERRARI and the original ReCrash tool leverage the `java.lang.instrument` API for load-time instrumentation), which primarily affects the initial program execution phase.

8.1 Naive versus Efficient Calling Context Reification

Table 1 compares the overhead of naive calling context reification using a thread-local variable ('Naive' columns) with the overhead due to our approach passing the calling context as extra method arguments ('Efficient' columns). The corresponding aspects, `CCAspectSlow` and `CCAspectFast`, are given in Figure 1 and in Figure 2. Both aspects were woven into all executing methods (in application classes as well as in the Java class library). While `CCAspectSlow` was woven with FERRARI and the AJ-UDI, `CCAspectFast` was woven with FERRARI and the AJ-CC-UDI, and the `CCAspectFast` itself was transformed with the AJ-CC-Trans tool.

We evaluated two different shadow stack variations. The first shadow stack stores *dynamic join points* (`JoinPoint` instances obtained in the aspects via `thisJoinPoint`). It exactly corresponds to the aspects in Figure 1 and in Figure 2. The second shadow stack holds *static join points* (`JoinPoint.StaticPart` instances obtained in the aspects via `thisJoinPointStaticPart`). While

[8]`http://groups.csail.mit.edu/pag/reCrash/`
[9]`http://www.spec.org/osg/jvm98/`

	Orig.	Dynamic Join Points				Static Join Points			
		Naive		Efficient		Naive		Efficient	
Client	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
compress	5.73	97.39	**17.00**	52.13	**9.10**	73.37	**12.80**	18.76	**3.27**
jess	1.46	41.02	**28.10**	20.79	**14.24**	32.06	**21.96**	7.63	**5.23**
db	14.14	79.08	**5.59**	48.63	**3.44**	62.88	**4.45**	20.72	**1.47**
javac	3.95	44.82	**11.35**	26.50	**6.71**	34.82	**8.82**	11.84	**3.00**
mpegaudio	2.47	45.19	**18.30**	22.60	**9.15**	34.52	**13.98**	7.55	**3.06**
mtrt	1.15	97.43	**84.72**	41.79	**36.34**	78.78	**68.50**	12.29	**10.69**
jack	3.48	27.63	**7.94**	17.96	**5.16**	21.75	**6.25**	8.51	**2.45**
Geo.mean	3.34	55.98	**16.76**	30.34	**9.08**	43.71	**13.09**	11.57	**3.46**
Server	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
compress	5.68	53.46	**9.41**	31.88	**5.61**	35.78	**6.30**	8.94	**1.57**
jess	1.47	20.29	**13.80**	12.36	**8.41**	14.94	**10.16**	3.16	**2.15**
db	13.71	37.67	**2.75**	33.81	**2.47**	30.14	**2.20**	15.21	**1.11**
javac	3.79	27.55	**7.27**	18.39	**4.85**	21.1	**5.57**	7.45	**1.97**
mpegaudio	2.48	25.61	**10.33**	13.09	**5.28**	17.48	**7.05**	3.89	**1.57**
mtrt	1.16	49.82	**42.95**	20.72	**17.86**	34.08	**29.38**	3.40	**2.93**
jack	3.48	18.15	**5.22**	11.98	**3.44**	13.53	**3.89**	5.58	**1.60**
Geo.mean	3.31	30.76	**9.29**	18.69	**5.65**	22.30	**6.74**	5.86	**1.77**

Table 1: Overhead comparison: CCAspectSlow (naive) versus CCAspectFast (efficient).

	Orig.	ReCrash App.		ReCrashAspect App.		ReCrashAspect App.+JDK	
Client	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
compress	5.73	138.84	**24.23**	43.05	**7.51**	51.09	**8.92**
jess	1.46	43.65	**29.90**	14.15	**9.69**	17.96	**12.30**
db	14.14	15.58	**1.10**	15.09	**1.07**	46.80	**3.31**
javac	3.95	17.07	**4.32**	13.03	**3.30**	23.53	**5.96**
mpegaudio	2.47	32.82	**13.29**	21.00	**8.50**	21.45	**8.68**
mtrt	1.15	49.79	**43.30**	34.06	**29.62**	41.04	**35.69**
jack	3.48	5.76	**1.66**	5.97	**1.72**	15.60	**4.48**
Geo.mean	3.34	28.47	**8.52**	17.54	**5.25**	28.11	**8.42**
Server	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
compress	5.68	82.58	**14.54**	27.24	**4.80**	28.82	**5.07**
jess	1.47	29.98	**20.39**	9.92	**6.75**	11.17	**7.60**
db	13.71	14.38	**1.05**	13.93	**1.02**	31.84	**2.32**
javac	3.79	12.71	**3.35**	11.78	**3.11**	16.83	**4.44**
mpegaudio	2.48	22.43	**9.04**	12.69	**5.12**	12.61	**5.08**
mtrt	1.16	29.90	**25.78**	19.73	**17.01**	19.69	**16.97**
jack	3.48	5.72	**1.64**	5.73	**1.65**	10.74	**3.09**
Geo.mean	3.31	20.89	**6.31**	13.03	**3.94**	17.28	**5.22**

Table 2: Overhead comparison: Original ReCrash tool versus ReCrashAspect.

dynamic join points provide more detailed calling context information, static join points cause much less overhead and provide enough information for building tools such as profilers.

Table 1 shows the measured execution times and the corresponding overhead factors ('ovh'). The 'Orig.' column is the execution time for the unmodified benchmarks.

For dynamic join points, our efficient calling context reification approach almost halves the overhead caused by the naive approach. In the Client VM, the overhead is reduced from factor 16.76 to factor 9.08 on average; in the Server VM, it is reduced from factor 9.29 to factor 5.65. In general, the overheads experienced in the Server VM are lower than in the Client VM, because the just-in-time compiler of the Server VM is known to perform more aggressive optimizations.

'mtrt', which is the most object-oriented benchmark in the JVM98 suite according to [11], suffers from the highest overhead. 'mtrt' invokes many methods with short bodies, resulting in excessive overhead due to the frequent advice invocations, the creation of the dynamic join points, and the updates of the shadow stack. For 'mtrt', our approach shows the highest overhead reductions (from factor 84.72 to 36.34 in the Client VM, respectively from factor 42.95 to 17.86 in the Server VM).

In contrast to dynamic join points that are created upon each advice invocation, static join points are created only once and stored in static fields. Thus, it can be expected that calling context reification using static join points causes significantly less overhead than using dynamic join points. Table 1 confirms this expectation. Because the high overhead of dynamic join point creation (which affects both the naive and the efficient approach) is avoided, the performance benefits of passing the calling context as extra arguments become more apparent. On average, the efficient scheme is almost 4 times faster than the naive approach (overhead reduction from factor 13.09 to 3.46 in the Client VM, respectively from factor 6.74 to 1.77 in the Server VM). Particularly impressive is the overhead reduction for 'mtrt', which exceeds factor 6 in the Client VM, respectively factor 10 in the Server VM.

Our approach to calling context reification benefits very much from typical compiler optimizations found in state-of-the-art JVMs, such as inlining and interprocedural register allocation. Figure 3 and Figure 4 show that the compiled advice methods of `CCAspectFast` are smaller than those of `CCAspectSlow`, easing inlining. Furthermore, since the reference to the object array `stack` is passed from caller to callee and is frequently accessed, it is a good candidate for allocation to a CPU register.

8.2 ReCrash versus ReCrashAspect

In our second evaluation, we compare the overhead of the original ReCrash tool with the overhead caused by the `ReCrashAspect` in Figure 10. For ReCrash, we use the most efficient mode, where only references to method arguments are kept on the shadow stack (no deep copying). This mode corresponds to the use of dynamic join points in the `ReCrashAspect`. The `ReCrashAspect` was woven by FERRARI and the AJ-CC-UDI, and transformed by the AJ-CC-Trans tool. We consider two different settings, 'App.' and 'App.+JDK'. In the former setting, the aspect is woven only into application classes, allowing a fair comparison with the original ReCrash tool. In the latter setting, the aspect is also woven into the Java class library, which is not supported by the original ReCrash tool.

Table 2 shows our measurements. Regarding the 'App.' settings, on average the `ReCrashAspect` is more than 60% faster than the original ReCrash tool. In contrast to the ReCrash tool, our aspect covers the execution of constructors and keeps the shadow stack consistent when an exception thrown in a callee is caught by a caller. Interestingly, for 'jack', the ReCrash tool slightly outperforms our aspect. 'jack' in known to be particularly "exception-intensive" [11]. Hence, our aspect incurs the overhead of shadow stack cleanup in exception handlers.

In the 'App.+JDK' setting, our aspect still slightly outperforms the original ReCrash tool on average. These results are surprising, because the original ReCrash tool uses a hand-crafted, low-level instrumentation and does not incur the overhead of invoking `aspectOf()` and the advice methods. Since our approach is portable and compatible with standard JVMs, we are able to leverage state-of-the-art compilation techniques that mitigate the overhead of advice method calls. We conclude that our high-level, AOP approach to the development of calling context sensitive tools can yield tools that outperform traditional implementations based on low-level bytecode instrumentation techniques, because we optimize the handling of calling context information.

9. RELATED WORK

Calling context reification can be achieved with standard AOP constructs. However, existing aspect weaving tools, such as AspectJ [18] and *abc* [6], do not support weaving in the Java class library, resulting in incomplete calling context information. Furthermore, such approaches can cause excessive overhead. Our approach solves these limitations thanks to FERRARI and the AJ-CC-UDI, which enable aspect weaving in the Java class library and optimize access to a complete calling context representation.

Calling context reification based on the `cflow` pointcut may introduce high overhead [5, 15]. The AspectJ and *abc* compilers rely on thread-local counters (instead of using a thread-local stack as in former AspectJ implementations) [18, 6] for the implementation of `cflow`. The *abc* compiler optimizes access to the thread-local counter within individual methods. Upon method entry, the thread-local variable is accessed only once and stored in a local variable. In contrast, our approach applies a global optimization by passing the reified calling context as extra arguments to all overloadable methods. *Nu* [12] uses an intermediate language approach to implement dynamic AOP constructs, such as `cflow`. More efficient implementations of `cflow` exploit direct access to JVM internals and can be integrated into the just-in-time compiler [9]; however, they require a modified JVM.

In Smalltalk, the call stack is directly accessible due to the reflective nature of the language. AspectS [16], an AOP framework implemented in Smalltalk, enables access to the call stack through the special variable `thisContext`. In Java, stack walking is used in JAsCo [27] and JBoss AOP [17] using the `Throwable` API, resulting in high overhead.

Steamloom [14, 8] and PROSE [24, 23] provide aspect support within the JVM, which may ease calling context reification thanks to the direct access to JVM internals. Steamloom is an extension of the Jikes RVM [2] supporting efficient aspect execution and dynamic aspect weaving. PROSE introduces aspect weaving through the Java Virtual Machine Debugger Interface (JVMDI), weaving at the just-in-time compiler level, and combines bytecode instrumentation with an extension of the Jikes RVM. These approaches trade portability for performance, since the aspect support is integrated in the JVM. Our approach supports aspect weaving within the Java class library in a portable manner and enables the use of state-of-the-art JVMs and aspect weaving tools, while providing complete calling context information in an efficient way.

Tracematches [1] is an extension of AspectJ enabling history-based programming to trigger the execution of extra code by specifying a pattern of events that cannot be expressed in AspectJ. For example, it is possible to trigger an event only upon a given sequence of method calls. Similar techniques are used in security tools for anomaly-based intrusion detection [13] that rely on calling context information. Our approach provides complete and customized calling context information enabling such functionality without extending the AOP language.

In addition to the shadow stack, our techniques can provide other calling context representations, such as the *calling context tree (CCT)* [3], which is commonly used for profiling and program analysis. Existing approaches that create accurate CCTs [25, 3] suffer from considerable overhead. Approaches based on sampling and stack-walking [29] help reduce the overhead, but at the expense of a loss of accuracy. Also the *probabilistic calling context (PCC)* [10] reduces the cost of computing calling context information. Unfortunately, these approaches do not create complete calling context representations and rely on native code, limiting portability. In contrast, our approach reconciles completeness of the calling context and moderate overhead, without resorting to JVM modifications.

10. CONCLUSION

In this paper we presented a customizable, efficient, accurate, and portable approach to calling context reification in Java and integrated it with AOP. Our approach enables rapid, AOP-based development of extensible and efficient profiling, debugging, and reverse engineering tools that require accurate calling context information as well as complete bytecode coverage.

As case study, we represented ReCrash as an aspect using a shadow stack; ReCrash is an existing tool for reproducing program failures. Using AOP, we resolved several limitations of ReCrash, such as inaccuracy and incompleteness of the shadow stack. Moreover, our approach significantly reduces the overhead of calling context reification.

Our work relies on FERRARI, a generic bytecode instrumentation framework, which allows user-defined instrumentation modules (UDIs) to instrument any code in a system having a bytecode representation. We implemented our instrumentation for calling context reification as a FERRARI UDI and composed it with the AspectJ weaver. The resulting composition of program transformations solves two limitations of current AOP environments, namely the impossibility to weave aspects into standard Java class libraries in a portable way, and the lack of an efficient mechanism to access complete calling context information.

Regarding ongoing research, we are using our techniques for an optimized implementation of the `cflow` pointcut. Moreover, we are integrating the customizable aspect compiler *abc* with FERRARI. We are extending the aspect language with low-level pointcuts at the bytecode and basic block levels, which we have already successfully used for specifying accurate and efficient cross-profilers for embedded Java systems.

Acknowledgements

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

11. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM.

[2] 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.

[3] 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.

[4] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In J. Vitek, editor, *ECOOP '08: Proceedings of the 22th European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 542–565, Paphos, Cyprus, 2008. Springer-Verlag.

[5] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni,

G. Sittampalam, and J. Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM.

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

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

[8] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 109–124, New York, NY, USA, 2006. ACM.

[9] C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini. Efficient control flow quantification. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 125–138, New York, NY, USA, 2006. ACM.

[10] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming, systems and applications*, pages 97–112, New York, NY, USA, 2007. ACM.

[11] J. Dujmovic and C. Herder. Visualization of Java workloads using ternary diagrams. *Software Engineering Notes*, 29(1):261–265, 2004.

[12] R. Dyer and H. Rajan. Nu: A dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, pages 191–202, New York, NY, USA, 2008. ACM.

[13] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[14] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An execution layer for aspect-oriented programming languages. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 142–152, New York, NY, USA, 2005. ACM.

[15] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, New York, NY, USA, 2004. ACM.

[16] R. Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[17] JBoss. Open source middleware software. Web pages at `http://labs.jboss.com/jbossaop/`.

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

[19] 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.

[20] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[21] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

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

[23] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, New York, NY, USA, 2003. ACM Press.

[24] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[25] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[26] Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. Web pages at `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, 2006.

[27] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[28] A. Villazón, W. Binder, and P. Moret. Aspect Weaving in Standard Java Class Libraries. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 159–167, New York, NY, USA, Sept. 2008. ACM.

[29] 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.