Comprehensive aspect weaving for Java

Alex Villazón *, Walter Binder, Philippe Moret, Danilo Ansaloni

Faculty of Informatics, University of Lugano, CH-6900 Lugano, Switzerland

A R T I C L E   I N F O

Article history:
Received 20 July 2009
Received in revised form 19 October 2009
Accepted 2 December 2009
Available online 29 April 2010

Keywords:
Aspect-oriented programming
Aspect weaving
Bytecode instrumentation
Profiling
Debugging
Detecting memory leaks
Recreating crashing conditions
Java Virtual Machine

A B S T R A C T

Aspect-oriented programming (AOP) has been successfully applied to application code thanks to techniques such as Java bytecode instrumentation. Unfortunately, with existing AOP frameworks for Java such as AspectJ, aspects cannot be woven into the standard Java class library. This restriction is particularly unfortunate for aspects that would benefit from comprehensive aspect weaving with complete method coverage, such as profiling or debugging aspects. In this article we present MAJOR, a new tool for comprehensive aspect weaving, which ensures that aspects are woven into all classes loaded in a Java Virtual Machine, including those in the standard Java class library. MAJOR includes the pluggable module CARAJillo, which supports efficient access to a complete and customizable calling context representation. We validate our approach with three case studies. Firstly, we weave existing profiling aspects with MAJOR which otherwise would generate incomplete profiles. Secondly, we introduce an aspect for memory leak detection that also benefits from comprehensive weaving. Thirdly, we present an aspect subsuming the functionality of ReCrash, an existing tool based on low-level bytecode instrumentation techniques that generates unit tests to reproduce program failures. Our aspect-based tools are concisely implemented in a few lines of code, and leverage MAJOR and CARAJillo for comprehensive aspect weaving and for efficient access to calling context information.

1. Introduction

Aspect-oriented programming (AOP) [30] is a powerful approach enabling a clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, monitoring and logging, and debugging support. AspectJ [29] provides AOP capabilities for Java, allowing new functionality to be systematically added to existing programs.

Java bytecode instrumentation is a widely used technique for profiling and, more generally, for any transformation in support of aspect weaving [18]. Based on bytecode instrumentation, aspect weavers, such as AspectJ [29] or abc [5], allow the insertion of code at well defined points in Java programs without resorting to source code manipulation.

Even though low-level instrumentation techniques have been successfully used to build tools for profiling, debugging, testing, and reverse engineering, the implementation of new tools using such techniques is tedious, error prone, and requires costly testing. We promote tool development in Java using high-level specifications of instrumentations with AOP, so as to hide low-level instrumentation details from the tool developer, easing maintenance and extension of tools.

Unfortunately, prevailing aspect weavers do not support comprehensive aspect weaving, notably because they prevent weaving in the Java class library. Hence, the applicability of AOP is limited, because tools such as profilers often require full...
method coverage. For example, when using standard AspectJ weavers, profiling aspects, such as DJProf [37], cannot profile method execution within the Java class library, resulting in incomplete profiles. This issue was reported as a major limitation of the approach by the authors of DJProf.

In addition, techniques for profiling, debugging, and reverse engineering often benefit from detailed calling context information [2,3]. As an example in the debugging area, calling context information has been used to reproduce the crashing conditions of a faulty application by storing copies of method arguments on a shadow stack [3]. Unfortunately, AspectJ does 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, thus limiting the applicability of AOP for profiling and debugging. Our work aims at filling this gap by enabling high-level instrumentation with full method coverage through comprehensive aspect weaving and efficient access to complete calling context information.

In general, instrumentation of the standard Java class library is difficult and may crash the JVM, because of its sensitivity to modifications, notably during JVM bootstrapping. While aspect support can be integrated at the JVM level [38], such an approach is restrictive because it prevents the reuse of existing AOP tools. In addition, using modified JVMs and native code makes it difficult to leverage standard, state-of-the-art JVM technologies.

In prior work we introduced FERRARI (Framework for Exhaustive Rewriting and Reification with Advanced Runtime Instrumentation) [7], a generic bytecode instrumentation framework supporting the instrumentation of the whole Java class library, including all core classes, as well as dynamically loaded classes. FERRARI provides a flexible interface enabling different user-defined instrumentations (UDIs) written in pure Java to control the instrumentation process.

In this article, we present a new framework for comprehensive aspect weaving based on FERRARI. Fig. 1 illustrates the architecture of the framework. Different aspect-based tools can be implemented using MAJOR,¹ which ensures comprehensive aspect weaving. MAJOR relies on AspectJ UDI (AJ-UDI), which adapts an existing and unmodified aspect weaver so as to weave aspects also into the Java class library. MAJOR features CARAJillo,² a pluggable module that combines aspect weaving with calling-context reification provided by the CC-UDI. The framework absolves the developer from dealing with low-level instrumentation details, which are handled by the standard AspectJ weaver in conjunction with FERRARI, MAJOR, and CARAJillo.

We show the soundness and potential of our approach with the following three case studies in the areas of profiling and debugging:

- We weave existing DJProf profiling aspects [37] with MAJOR in order to generate complete profiles covering all methods executed in the JVM. If woven with prevailing aspect weavers, DJProf aspects generate incomplete profiles [37].
- We introduce an aspect-based memory leak detection tool, which is able to generate a complete list of allocated objects that could lead to a memory leak. Memory leaks often occur due to containers or collection data structures that refer to unused data, which are typically allocated within the Java class library, and therefore cannot be tracked if the aspect is woven with prevailing aspect weavers.
- We present an aspect that subsumes the functionality of ReCrash. ReCrash is an existing tool that generates unit tests to reproduce program failures. ReCrash relies on low-level bytecode instrumentation and uses a shadow stack so as to keep information on an eventual crash [3]. The aspect offers several advantages compared to the original ReCrash tool, such as improved functionality, increased coverage (supporting developers when debugging the Java class library), and improved performance.

Our aspect-based tools are concisely implemented in a few lines of code and leverage MAJOR for comprehensive aspect weaving and CARAJillo for efficient access to calling context information.

¹ MAJOR stands for MAJOR is AspectJ with Overall Rewriting.
² CARAJillo stands for Context Aware Rewriting for AspectJ. Carajillo is a Spanish coffee ‘enhanced’ with a strong distilled beverage.
This article refines and extends the concepts introduced in [47]. As contributions, in this article we highlight the benefits of comprehensive aspect weaving and describe the techniques underlying MAJOR and CARAJillo. We show that our approach is compatible with the standard AspectJ, and we present concrete examples of existing and new aspect-based tools that benefit from comprehensive aspect weaving.

This article is structured as follows. Section 2 summarizes the features provided by FERRARI, our generic instrumentation framework. Section 3 introduces basic concepts of aspect weaving and discusses the intricacies of weaving the Java class library. Section 4 discusses the implementation of the AJ-UDI and how we adapt the AspectJ weaver to support comprehensive aspect weaving. Section 5 presents our approach to efficient calling context reification and the implementation of the CC-UDI. Sections 6, 7 and 8 describe and evaluate the aspect-based tools of our case studies. Section 9 addresses related work. Finally, Section 10 concludes this article.

2. Instrumentation framework

Our framework for comprehensive aspect weaving relies on FERRARI [7], a generic bytecode instrumentation framework supporting the instrumentation of all classes loaded in a JVM. FERRARI is neither an application-specific framework nor a low-level bytecode instrumentation toolkit. Instead, it generates the necessary program logic to enable custom instrumentation of the Java class library and of application classes. FERRARI solves the problem of bootstrapping the JVM with an instrumented Java class library. It provides a flexible interface enabling different user-defined instrumentation modules (UDIs), which may be written in pure Java, to control the instrumentation process.

FERRARI consists of a static instrumentation tool and a runtime instrumentation agent. FERRARI defines an interface that the UDI has to implement and invokes the UDI through this interface. The UDI may change method bodies, add new methods (with minor restrictions), and add fields (with some restrictions). 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 provides a tool to statically instrument the Java class library according to a given UDI. Fig. 2(a) shows the static instrumentation of the Java class library (represented by rt.jar), resulting in the instrumented version INST_rt.jar that is used by the JVM executing the application under dynamic instrumentation (see Fig. 2(b)).

Application classes are dynamically instrumented by FERRARI's agent in collaboration with the UDI. The agent is based on the java.lang.instrument package introduced in Java 5, ensuring portability. Fig. 2(b) shows how application classes (App) are instrumented dynamically so that their instrumented versions (INST_App) are those actually linked by the JVM.

The current implementation of FERRARI uses Apache's bytecode engineering library BCEL [13]. UDIs are nevertheless free to use any bytecode engineering library, such as BCEL, ASM [36], Javassist [12], Soot [45], etc.

FERRARI offers generic mechanisms to ensure complete instrumentation coverage of any code in a system which has a corresponding bytecode representation. To this end, (1) it ensures that UDI-inserted code is not executed before the JVM has completed bootstrapping and (2) it provides support for temporarily bypassing the execution of inserted code for each thread during load-time instrumentation.

Regarding issue (1), FERRARI keeps a copy of the original code of every instrumented method and uses a global flag, the Bootstrap Inserted-code Bypass (BIB), to bypass the execution of UDI-inserted code during the bootstrapping of the JVM. For details, see [7].

Concerning issue (2), FERRARI introduces a thread-local flag, the Dynamic Inserted-code Bypass (DIB), which allows per-thread bypassing of UDI-inserted code. To this end, FERRARI inserts the boolean instance field dibFlag into the java.lang.Thread class. If the flag is set to true, the current thread bypasses UDI-inserted code. The dibFlag is exposed to the UDI developer through the DIB class (see Fig. 3).

---

3 We define the bootstrapping phase to last until FERRARI's instrumentation agent starts execution, which happens before the end of the JVM startup (see JVM Specification, Second Edition, Section 5.5) [33].
The bypasses induce some constraints for UDI development. UDI-inserted static or instance fields must be initialized to Java’s default values. Otherwise, the inserted code to initialize the added fields may be bypassed, resulting in incompletely initialized classes or objects. To mitigate this limitation, FERRARI offers special support for introducing extra classes that can hold added static fields [7]. This support is only available for the static instrumentation of the Java class library.

In general, UDI-inserted code often introduces dependencies on UDI-specific runtime classes (we call them “UDI-runtime-classes”). The UDI developer must ensure that methods in UDI-runtime-classes do not execute any instrumented code. To this end, the DIB mechanism can be used, as shown in Fig. 4, which allows the calling thread to temporarily bypass UDI-inserted code at runtime. This issue is particularly important in support of comprehensive aspect weaving and will be explained in Section 3.2.

When a new thread is created, it “inherits” the dibFlag value from the current thread. FERRARI modifies the constructors of java.lang.Thread accordingly. Consequently, if the UDI or UDI-runtime-classes spawn threads while the dibFlag is true, the new threads will also bypass UDI-inserted code. By default, the dibFlag is set to false and therefore instrumented code is executed when the BIB is disabled by FERRARI’s agent after bootstrapping. This ensures that UDI-inserted code is executed when the application’s main(...) method is invoked.

FERRARI classes, the instrumentation agent, the UDI, UDI-runtime-classes, and classes of the bytecode engineering library are excluded from normal instrumentation. In order to avoid name-clashes with application classes, these classes are loaded into a special classloader namespace. All other classes are instrumented either statically or dynamically at load-time.

In addition to instrumentation support for complete method coverage, in many cases it is desirable to track invocations to native methods which are not amenable to bytecode instrumentation. To this end, FERRARI uses native method prefixing, a feature of the standard JVM Tool Interface (JVMTI) [41] introduced in Java 6 that allows wrapping native methods with bytecode bodies, which are amenable to bytecode instrumentation.

3. Comprehensive aspect weaving

In order to better understand how comprehensive aspect weaving is achieved, we first give some background information of the most important concepts of AOP and aspect weaving. Then we describe the intricacies of weaving the Java class library.

3.1. AOP

In AspectJ, an aspect is an extended class with additional constructs. A join point is any identifiable execution point in a system (e.g., method call, method execution, object construction, variable assignment, etc.). Join points are the places where a crosscutting action can be inserted. The user can specify weaving rules to be applied to join points through so-called pointcuts and advice. A pointcut identifies or captures join points in the program flow, and advice are the actions to be applied.

4 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.
import java.lang.ref.*;  
import java.util.*;

public aspect ObjectTracker {  
    static private final Set<WeakReference> refs =  
        Collections.synchronizedSet(new HashSet<WeakReference>();  
    static private final ReferenceQueue<Object> refQueue = new ReferenceQueue<Object>();

    pointcut allAllocs() : call(*.new(..)) && !within(ObjectTracker);  
    after() returning(Object o) : allAllocs() {  
        WeakReference objRef = new WeakReference(o, refQueue);  
        refs.add(objRef);  
    }  
    ...
}

Fig. 5. Aspect tracking object allocations.

AspectJ supports three kinds of advice: before, after, and around, which are executed prior, following, or surrounding a join point’s execution. Aspects are compiled into standard Java classes. In the aspect class, advice are compiled into methods. During the weaving process, the weaver inserts code in the woven class to invoke these advice methods. Advice can receive some context information, e.g., to identify which join point has been captured.

During the execution of a woven class, by default, a singleton instance of the aspect is instantiated. Several aspects can be woven simultaneously and can therefore coexist during the execution.

3.2. Intricacies of weaving the Java class library

Let us consider a simple profiling aspect. Fig. 5 shows the ObjectTracker aspect that captures every object allocation by intercepting constructor calls (specified by call(*.new(..)) in the allAllocs pointcut). After each object allocation, a weak reference to the allocated object is stored in a set. The set must be updated atomically, as a single instance of the aspect is shared by all executing threads. The !within() pointcut designator prevents the aspect itself to be woven in order to avoid infinite recursions. The aspect may periodically analyze the set refs and use Java’s shutdown hook mechanism to dump the final results upon application termination (not shown in Fig. 5).

The code of the ObjectTracker aspect could be part of a memory leak profiling tool, where it is essential to keep track of memory allocations also in the Java class library. This is because many memory leaks occur in containers that reference unused data entries [23,50], and usually the standard Java collection classes are used as containers. Unfortunately, when woven with prevailing AspectJ weavers, the aspect will track only allocations in application code. Consequently, without comprehensive aspect weaving, AOP is not well suited for building sophisticated profiling and debugging tools, such as the container-based heap tracking profiler described in [50]. Our approach aims at filling this gap so as to enable comprehensive aspect weaving, while ensuring portability and compatibility with standard AspectJ.

There are two main issues related to the weaving of the Java class library. The first issue concerns the sensitivity to modifications of core Java classes, notably during JVM bootstrapping. Because the weaving process introduces new dependencies between the woven classes and the aspects, loading a woven core Java class, such as java.lang.Object, may break bootstrapping and eventually crash the JVM. To solve this issue, we rely on FERRARI’s support for bootstrapping the JVM with an instrumented Java class library.

The second issue is related to advice execution during the execution of woven code. In the example of the ObjectTracker aspect in Fig. 5, the advice body allocates an instance of the WeakReference class, which is part of the Java class library. As shown in Fig. 6(a), if the aspect is woven into both the application code and the Java class library using prevailing aspect weavers (assuming that the bootstrapping problem is solved), an infinite recursion will happen, because the invocation of woven code by the advice will recursively trigger the invocation of the advice (which captures also the WeakReference allocation).

To solve this issue, MAJOR uses FERRARI’s DIB mechanism. The method bodies are duplicated such that the original and the instrumented versions of the bytecode are kept together, and a conditional selects the version to be executed upon method entry depending on the state of the DIB. This solution is illustrated in Fig. 6(b). The pseudo-code in Fig. 7 shows how the code pattern to activate the DIB (see Fig. 4) is applied to advice execution.

4. The AspectJ UDI (AJ-UDI)

In this section we discuss how we adapted an existing AspectJ weaver as a FERRARI UDI. No modifications to AspectJ were necessary. Below, we describe different transformations made during the weaving process and their relation to FERRARI features, which are fundamental to support comprehensive aspect weaving, and then we describe the implementation of

---

5 The WeakReference type does not prevent the referenced object from being garbage collected. The ReferenceQueue instance is used by the garbage collector to place weak references that are cleared.
the AJ-UDI and discuss its limitations. In this section we refer to the bytecode-level instrumentation made by the AspectJ weaver [24].

4.1. Aspect weaving

In the aspect class, advice are compiled into methods. The weaver inserts code to invoke these advice methods in the woven class. Aspect classes are used by the AJ-UDI to weave both application classes and the Java class library. Since advice methods are invoked by instrumented code, aspect classes are UDI-runtime-classes. Hence, the advice methods must not execute any instrumented code.

The execution of advice bodies implies the creation of an aspect instance. By default, there is a singleton aspect instance that is accessed by the woven classes through the static method aspectOf() that is generated by the aspect compiler in the aspect class. The AJ-UDI must ensure that the advice methods and the method aspectOf() do not execute any instrumented code in order to prevent infinite recursions. To temporarily bypass the execution of instrumented code, the developer may use the DIB API, or alternatively use an automated tool (called AspectTransformer), which applies the code pattern shown in Fig. 4 to every constructor, static initializer, advice method, and generated method in the aspect class.

Each kind of join point has its own corresponding bytecode representation (e.g., method execution – entire code segment of method body; field get – getfield or getstatic; etc.). Thus, the weaver analyzes and modifies the bytecode (1) to retrieve the aspect instance (call to aspectOf() in the aspect class) and (2) to invoke the advice methods, before, after, or around the corresponding bytecode representation of the join point.

If an advice makes use of AspectJ’s reflective API to access static information about the join point (e.g., through the JoinPoint.StaticPart interface or when using the thisJoinPointStaticPart pseudo-variable), the weaver adds
static fields in the woven class to store the corresponding references and modifies the static initializer. As described in Section 2, FERRARI offers support for UDI to handle added static fields through extra classes. The AJ-UDI uses this feature for weaving the Java class library.

In addition to the default singleton aspect instances, AspectJ also supports non-singleton ones through special per* clauses (e.g., for per-object or per-control flow aspect associations). Unfortunately, this mechanism modifies the class hierarchy of the woven classes (to implement a compiler generated interface). Changing the hierarchy of classes from the Java class library may break the bootstrapping and, therefore, the AJ-UDI does not support constructs that modify the class hierarchy for weaving the Java class library. Other transformations (e.g., method body modifications, insertion of advice invocation, additional static fields, access to reflective information within advice bodies) are supported for comprehensive aspect weaving.

All the previously described weaving mechanism and code transformations, called dynamic-crosscutting, are not controlled by the aspect code, but are specific to the implementation of the weaver. AspectJ also supports static-crosscutting that enables structural transformations of types (classes, interfaces, and aspects) directly within the aspect description. The inter-type declaration mechanism allows adding static or instance fields, methods, and also modifying the class hierarchy with some restrictions. For weaving the Java class library, the AJ-UDI does not support static-crosscutting that modifies the class hierarchy.

4.2. Adapting the AspectJ weaver

AspectJ supports compile-time and load-time weaving. The ajc compile-time tool is a compiler and bytecode weaver, i.e., it compiles and weaves applications and aspects from source code and can also weave aspects directly from bytecode. The ajc tool is based on an extension of the Eclipse Java compiler and an extension of Apache BCEL for bytecode weaving. AspectJ supports two different approaches for load-time weaving: the first one uses a customized class loader and the second one is based on the java.lang.instrument API.

For the Java class library, we use a two-phase instrumentation: first we use the ajc bytecode weaver to weave the aspect into the Java class library (see Fig. 8 on the left side), and in the second phase, the AJ-UDI uses the woven Java class library such that FERRARI can insert the bypasses (see Fig. 8 on the right side). Note that in the first phase, even though ajc weaves aspects into the Java class library, the resulting code is unusable, as the standard aspect weaver does not handle the issues related to bootstrapping and to infinite recursions described in Section 3.2.

During the second phase, since no method signatures are modified by the weaver, the AJ-UDI performs a rather simple sequence: For each class to instrument, it reads the already woven version (rather than invoking the weaver) and compares it with the original class. It determines which methods were modified by the weaver, and checks if static fields were added (e.g., static fields related to AspectJ’s reflective API described before). The AJ-UDI moves added static fields and the corresponding parts of the static initializers into extra classes. This information is provided to FERRARI. The resulting INST_rt.jar in Fig. 8 has the woven classes with the bypasses together with the extra classes.

AspectJ provides a WeavingAdaptor class allowing third party applications to interact with the weaver. The adaptor receives a class as a byte array and returns the woven class also as a byte array. The AJ-UDI uses the WeavingAdaptor for dynamic instrumentation. FERRARI’s runtime instrumentation agent plays a similar role to AspectJ’s load-time mechanisms. Once a class is woven, the AJ-UDI determines which methods were modified by the weaver, and tells FERRARI to generate bypass code to enable callbacks from advice bodies into application code.

Fig. 9 illustrates the execution of a dynamically woven application running on top of a woven Java class library. In this example we assume that the same aspect was woven in the application and in the Java class library (which is not necessarily

---

6 Thanks to the JVM's lazy class initialization (see JVM Specification, Second Edition, Section 5.5) [33], the extra classes will be initialized after bootstrapping, because they are only used by UDI-inserted code, which is guaranteed not to execute during bootstrapping. Since FERRARI does not insert bypasses in the static initializers of extra classes, the UDI-inserted static fields in the extra classes will be properly initialized.
always the case, as different aspects can be woven into the Java class library and the application). Both the woven Java class library and application code (INST_rt.jar and INST_App) invoke the advice in the aspect class, i.e., the UDI-runtime-class, which uses AspectJ’s runtime system for full dynamic aspect support. Thus, FERRARI and the AJ-UDI provide advanced support for weaving the Java class library which is complementary to the features offered by existing AspectJ compilers and weavers.

5. Efficient calling context reification

Comprehensive aspect weaving enables the rapid prototyping of profiling tools, while ensuring portability. However, comprehensive aspect weaving is not enough to implement more sophisticated profiling and debugging tools. In this section we discuss a naive approach to calling context reification and compare it to our approach to efficient calling context reification, which is complementary to comprehensive aspect weaving for building efficient tools.

5.1. Approaches to calling context reification

Existing techniques for profiling and debugging often benefit from detailed calling context information. For example, the Calling Context Tree (CCT) [2] is a popular profiling datastructure that helps locate performance bottlenecks in programs. As another 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 [3].

Unfortunately, existing AOP languages, such as AspectJ, 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 or debugging.

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. AspectJ provides limited access to dynamic and static calling context information through its reflective API. For example, advice bodies can access limited static information about the captured joinpoint through the thisJoinPointStaticPart pseudo-variable. However, there is no efficient, built-in construct to access complete calling context.

MAJOR features efficient calling context reification through the pluggable module CARAjillo, which allows aspects to access complete and accurate calling context information. The calling context information is passed as extra arguments to every method, including advice methods. To this end, in addition to the transformations performed by the AJ-UDI for comprehensive aspect weaving, CARAjillo triggers another transformation, so as to extend the method signatures and to create wrapper methods to handle callbacks from native code, which is not aware of the extra arguments. This transformation is performed by another FERRARI UDI, the calling context reification UDI (CC-UDI).

5.2. Naive calling context reification using a thread-local variable

While it is possible to rely on standard AspectJ constructs to reify the calling context, such a naive approach suffers from several drawbacks. To illustrate this, let us consider the aspect CCAspectSlow in Fig. 10 that maintains a shadow stack for each thread. To simplify the discussion, we do not consider execution of constructors, but only method executions. 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. To represent the method invocations on the shadow stack, we use static joinpoints, which convey the signatures of the invoked methods.

The aspect causes high runtime overhead because the thread-local variable TL is accessed in each advice body. 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.

This problem 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. 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 AJ-UDI.

5.3 Efficient calling context reification with extra method arguments

Fig. 11 shows an aspect that efficiently reifies the call stack. Upon method entry, the aspect pushes the context information of the callee (through the thisJoinPointStaticPart pseudo-variable) onto a shadow stack. The calling context representation is configurable and is exposed to the aspect through marker methods. Marker methods are actually never invoked (highlighted by the fact that they throw UnsupportedOperationException), but only used to ensure compatibility with the standard AspectJ compiler. The CC-UDI transforms the advice signature to pass the calling context as extra arguments, and the bytecodes corresponding to the invocation to the marked methods are replaced with bytecodes to access the extra arguments. In the example, thisStack() gives access to a thread-confined shadow stack (an array of objects), and thisSP() corresponds to the current stack pointer as an integer value. The CC-UDI takes care of correctly updating the value of the stack pointer, before passing it as extra argument. Upon method return, the calling context is popped off the shadow stack by nullifying the current position.

The CC-UDI supports a configurable number of extra arguments with different types. It uses configuration parameters accessContext, which refer to the static marker methods to access the ith extra argument within aspects. In the case of the shadow stack, accessContext1 = thisStack() and accessContext2 = thisSP(). More details about the configuration and the intricacies of introducing the reified calling context as extra arguments can be found in [48].

Thanks to our approach to comprehensive aspect weaving, the generated shadow stack corresponds exactly to the execution call stack. Without MAJOR and CARAJIllo, the only means at hand to gather equivalent information are the Throwable API (which causes high overhead), JVMTI native code agents to capture every method entry and completion (which also causes high overhead, as the corresponding JVMTI events prevent just-in-time compilation), or a modified JVM (compromising portability).
Fig. 11. Simplified aspect for maintaining a shadow stack.

Fig. 12. Overview of the transformation steps.

5.4. Composition of transformations for CARAJillo

CARAJillo combines the transformations done by the AJ-UDI described in Section 4 with those of the CC-UDI. We call this composition the AJ-CC-UDI.

Fig. 12 gives an overview of the transformation steps. The aspects are compiled using AspectJ’s ajc compiler. The compiled aspects are then passed to the CC-UDI, which extends the method signatures so as to pass the calling context information as extra arguments. The transformed code is then handed to FERRARI, which generated the bypasses to skip UDI-inserted code during JVM bootstrapping, during load-time instrumentation, and when executing methods of the aspect.

The compiled aspect (which is not processed by FERRARI) is transformed by the automated tool AspectTransformer in order to handle the reified calling context. AspectTransformer 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. 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, 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 advice bodies 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.
5.5. Naive versus efficient calling context reification

To evaluate the efficiency of our approach, we use the standard SPEC JVM98 [43] benchmark suite to measure the overhead and compare it to the naive approach. Our test platform is a Linux Fedora Core 2 computer (Intel Pentium 4, 2.66 GHz, 1024 MB RAM). The metric used for JVM98 is the execution time in seconds. We present measurements made with the Sun JDK 1.7.0-ea-b25 HotSpot VMs (Client and Server VMs). In order to attenuate the perturbations due to class-loading, load-time weaving, and just-in-time compilation, we report the median of 15 runs within the same JVM process. We also report average execution time and overhead factor for the benchmark suite, which we compute as the geometric mean ("Geo.mean"). All our evaluations in this article were made with MAJOR (version 0.5) and AspectJ (version 1.6.2).[8]

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 Fig. 10 and in Fig. 11. Both aspects were woven into all executing methods (in application classes as well as in the Java class library). While CCAspectSlow was woven with MAJOR, CCAspectFast was woven with CARAJillo enabled, and the CCAspectFast itself was transformed with the AspectTransformer tool.

We evaluated two different shadow stack variations. The first shadow stack stores dynamic join points (JoinPoint instances obtained in the aspects via thisJoinPoint). The second shadow stack holds static join points (JoinPoint, StaticPart instances obtained in the aspects via thisJoinPointStaticPart); it exactly corresponds to the aspects in Fig. 10 and in Fig. 11. While dynamic join points provide more detailed calling context information, static join points cause much less overhead and usually provide enough information for building tools such as profilers [6].

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 [17], 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

---

7 MAJOR has been successfully tested in different platforms (Windows, Linux, MacOSX) with different state-of-the-art JVMs, such as Sun’s Hotspot VMs or IBM’s J9 VMs.
9 http://www.eclipse.org/aspectj/.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Overhead comparison: CCAspectSlow (naive) versus CCAspectFast (efficient).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dynamic join points</td>
</tr>
<tr>
<td></td>
<td>Orig.</td>
</tr>
<tr>
<td>Client</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[s]</td>
</tr>
<tr>
<td>compress</td>
<td>5.73</td>
</tr>
<tr>
<td>jess</td>
<td>1.46</td>
</tr>
<tr>
<td>db</td>
<td>14.14</td>
</tr>
<tr>
<td>javac</td>
<td>3.95</td>
</tr>
<tr>
<td>mpegudio</td>
<td>2.47</td>
</tr>
<tr>
<td>mtrt</td>
<td>1.15</td>
</tr>
<tr>
<td>jack</td>
<td>3.48</td>
</tr>
<tr>
<td>Geo.mean</td>
<td>3.34</td>
</tr>
<tr>
<td>Server</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[s]</td>
</tr>
<tr>
<td>compress</td>
<td>5.68</td>
</tr>
<tr>
<td>jess</td>
<td>1.47</td>
</tr>
<tr>
<td>db</td>
<td>13.71</td>
</tr>
<tr>
<td>javac</td>
<td>3.79</td>
</tr>
<tr>
<td>mpegudio</td>
<td>2.48</td>
</tr>
<tr>
<td>mtrt</td>
<td>1.16</td>
</tr>
<tr>
<td>jack</td>
<td>3.48</td>
</tr>
<tr>
<td>Geo.mean</td>
<td>3.31</td>
</tr>
</tbody>
</table>
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. This confirms the effectiveness of our approach.

6. Case study 1: applying existing profiling aspects

To show the applicability to existing aspects, we evaluated MAJOR using DJProf [37], a set of aspects for profiling heap usage, average object lifetime, average object “waste” time (object lifetime minus its useful time), time spent in each method invocation, and the number of method calls. Below we make reference to these aspects as heap, lifetime, waste, cpu, and call-count. As the authors of DJProf witnessed, the major limitation of DJProf was the inability to weave the profiling aspects into the Java class library, resulting in incomplete profiles. Our approach to comprehensive aspect weaving solves exactly that problem.

In the DJProf aspects (DJProf version 1.0)\(^\text{10}\) we corrected some race conditions (e.g., by using atomic integers for counters that are shared between threads), but we did not change the structure of the aspects, since our goal is to show that our framework works well with existing aspects. We used our AspectTransformer tool to add the DIB to every advice invoking instrumented code.

6.1. Evaluation

For our evaluation, we use two benchmark suites, SPEC JVM98 [43] (problem size 100) and DaCapo [9] (version ‘dacapo-2006-10-MR2’; default workload size). Our test platform is the same as in the evaluation in Section 5.5. We also report the median of 15 runs, and the average execution time, computed as the geometric mean.

Although we successfully applied the waste aspect to all benchmarks, we excluded waste from our evaluation, because of its excessive overhead [37]. Alternative and more efficient implementations of this aspect are available in DJProf, but unfortunately they modify the class hierarchy, preventing weaving of the Java class library.

The lifetime and cpu profiling aspects use a configurable sampling mechanism to reduce overhead. In our configuration, only every 100th object is profiled with the lifetime aspect, and the sampling interval is set to 100 ms for the cpu aspect.

Tables 2 and 3 show our measurements and the overhead for the different profiling aspect. In the setting ‘AspectJ’ we are using the original DJProf aspects woven by the AspectJ load-time weave agent (running with the original Java class library).

---

10 http://ecs.vuw.ac.nz/~djp/djprof/.
potential memory leaks if object allocation is implicit (no new objects, and method invocations passing objects across package boundaries). Unfortunately, such an approach fails to capture the application code is woven with an aspect capturing constructor calls, field assignments to track references between the JVM.

Information about loitering objectson the heap. Such profilers use an native profiling API of the underlying JVM, such as object retention. Detecting memory leaks can be difficult; typically, developers have to use profilers for digging out information about loitering objects on the heap. Such profilers use a native profiling API of the underlying JVM, such as the JVMPI \([14]\) or the more recent JVMTI \([11]\), or rely on a modified JVM \([11,14]\).

Some attempts have been made to build memory leak detection tools based on aspects \([31]\). In such an approach, the application code is woven with an aspect capturing constructor calls, field assignments to track references between objects, and method invocations passing objects across package boundaries. Unfortunately, such an approach fails to capture potential memory leaks if object allocation is implicit (no new instruction) in the application code as is the case in the code

whereas the setting ‘MAJOR’ corresponds to the corrected thread-safe DJProf aspects woven with MAJOR (running with the woven Java class library). We present the execution times for the various settings and the corresponding overhead factors (‘ovh’). The ‘Orig.’ column is the execution time for the unmodified benchmarks.

For the aspects heap, lifetime, and cpu, we observe rather uniform overhead. One exception is ‘mtrt’, where the overhead caused by the cpu aspect when weaving the Java class library reaches a factor of 22.09, even though the aspect uses sampling to reduce the overhead. The reason for the excessive overhead is that the cpu aspect intercepts both method execution and method calls and invokes advice before and after each of them. This causes high overhead for applications with many invocations of short methods, as is the case for ‘mtrt’ \([17]\).

For call-count, the overhead is higher because this aspect captures all method calls (no sampling). We observe a high overhead for ‘mtrt’ in the ‘AspectJ’ setting (factor 25.41), because the call-count aspect updates a global structure each time the advice is invoked. For the ‘MAJOR’ setting, the overhead reaches a factor of 41.45. Even though ‘mtrt’ is a not a Java class library intensive benchmark \([8,35]\), the overhead can be explained by the frequent activation of the DIB within advice bodies (required to update more complex data structures to ensure thread-safety), which is not present in the original DJProf profilers corresponding to the ‘AspectJ’ setting.

### 7. Case study 2: memory leak detection

In this second case study, we describe an aspect taking advantage of the comprehensive aspect weaving of MAJOR which enables simple leak analysis without resorting to low-level instrumentation mechanism. Here we do not use CARAJillo, because our evaluation concentrates on the effectiveness of our approach in detecting potential memory leaks, but not on the performance of the leak detector.

#### 7.1. Memory leaks

Despite automated memory management, Java programs can still suffer from memory leaks resulting from unintentional object retention. Detecting memory leaks can be difficult; typically, developers have to use profilers for digging out information about loitering objects on the heap. Such profilers use a native profiling API of the underlying JVM, such as the JVMPI \([40]\) or the more recent JVMTI \([41]\), or rely on a modified JVM \([28,14]\).

Some attempts have been made to build memory leak detection tools based on aspects \([31]\). In such an approach, the application code is woven with an aspect capturing constructor calls, field assignments to track references between objects, and method invocations passing objects across package boundaries. Unfortunately, such an approach fails to capture potential memory leaks if object allocation is implicit (no new instruction) in the application code as is the case in the code

### Table 3

<table>
<thead>
<tr>
<th>DaCapo</th>
<th>Orig. [s]</th>
<th>'cpu' aspect [s]</th>
<th>MAJOR 'ovh' [s]</th>
<th>'call-count' aspect [s]</th>
<th>MAJOR 'ovh' [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>3.31</td>
<td>6.77</td>
<td>2.05</td>
<td>19.60</td>
<td>5.92</td>
</tr>
<tr>
<td>bloat</td>
<td>13.79</td>
<td>23.57</td>
<td>1.71</td>
<td>115.50</td>
<td>8.38</td>
</tr>
<tr>
<td>chart</td>
<td>11.35</td>
<td>17.61</td>
<td>1.55</td>
<td>52.24</td>
<td>4.60</td>
</tr>
<tr>
<td>eclipse</td>
<td>59.44</td>
<td>66.28</td>
<td>1.12</td>
<td>166.46</td>
<td>2.80</td>
</tr>
<tr>
<td>fop</td>
<td>2.15</td>
<td>2.95</td>
<td>1.37</td>
<td>7.06</td>
<td>3.28</td>
</tr>
<tr>
<td>hsqldb</td>
<td>7.59</td>
<td>13.35</td>
<td>1.76</td>
<td>31.16</td>
<td>4.11</td>
</tr>
<tr>
<td>jython</td>
<td>9.15</td>
<td>40.59</td>
<td>4.44</td>
<td>81.60</td>
<td>8.92</td>
</tr>
<tr>
<td>luindex</td>
<td>15.94</td>
<td>53.20</td>
<td>3.34</td>
<td>91.10</td>
<td>5.72</td>
</tr>
<tr>
<td>lusearch</td>
<td>18.84</td>
<td>39.99</td>
<td>2.12</td>
<td>78.26</td>
<td>4.15</td>
</tr>
<tr>
<td>pmd</td>
<td>11.95</td>
<td>16.96</td>
<td>1.42</td>
<td>53.69</td>
<td>4.49</td>
</tr>
<tr>
<td>xalan</td>
<td>18.82</td>
<td>31.77</td>
<td>1.69</td>
<td>95.90</td>
<td>5.10</td>
</tr>
<tr>
<td>Geo.mean</td>
<td>11.09</td>
<td>20.95</td>
<td>1.89</td>
<td>56.64</td>
<td>4.93</td>
</tr>
</tbody>
</table>

### JVM98

| compress | 5.73 | 11.48 | 2.00 | 31.52 | 5.50 | 28.24 | 4.93 | 44.28 | 7.73 |
| jess     | 1.46 | 3.91  | 2.68 | 12.30 | 8.42 | 12.31 | 8.43 | 21.78 | 14.92 |
| db       | 14.14| 15.23 | 1.08 | 29.92 | 2.12 | 14.53 | 1.03 | 44.98 | 3.18 |
| javac    | 3.95 | 7.30  | 1.85 | 16.69 | 4.23 | 11.90 | 3.01 | 25.88 | 6.55 |
| mpegaudio| 2.47 | 6.24  | 2.53 | 14.90 | 6.03 | 14.84 | 6.01 | 21.62 | 8.75 |
| mtrt     | 1.15 | 8.80  | 7.65 | 25.40 | 22.09| 29.22 | 25.41| 47.69 | 41.45 |
| jack     | 3.46 | 4.71  | 1.35 | 11.14 | 3.20 | 4.94  | 1.42 | 16.57 | 4.76 |
| Geo.mean | 3.34 | 7.46  | 2.24 | 18.75 | 5.62 | 14.41 | 4.32 | 29.44 | 8.82 |
Fig. 13. Example of a memory leak in Java.

excert

In this example, the method `slowlyLeakingVector(...)` simply adds and removes elements from a vector. However, the remove operation leaves objects in the vector which cannot be garbage collected, representing a potential memory leak (the allocation is actually performed by the `java.lang.Integer` class and therefore cannot be intercepted at the application level). By enabling aspects to be woven into the Java class library, our approach solves this limitation allowing to identify such potential memory leaks.

In the following we present the `LeakDetectorAspect` and show how the memory leak in Fig. 13 is found.

7.2. The `LeakDetectorAspect` implementation

The main idea of the `LeakDetectorAspect` is to capture all object allocation sites and to keep a weak reference to every allocated object together with additional information about the allocation context. When the garbage collector reclaims unused objects, the weak references are removed. The remaining referenced objects correspond to potential memory leaks.

Fig. 14 shows a simplified version of the `LeakDetectorAspect`. The `allAllocs` pointcut (line 2) specifies that every object allocation join point (i.e., calls to `new`) must be captured.

The advice associated with the `allAllocs` pointcut is shown in lines 8–12. We use the `after returning` construct to get the reference to the newly created object, which is passed as the `newObject` argument to the advice. We create a weak reference to `newObject` with additional information about the allocation context. To this end, we use the `java.lang.ref` API which provides three reference types (`SoftReference`, `WeakReference`, and `PhantomReference`). These reference types do not prevent the referenced object from being garbage collected. We have chosen the `PhantomReference` type, because in contrast to the others, the referenced objects reclaimed by the garbage collector are “cleared” (i.e., set to `null`) only after finalization. This ensures that our aspect does not miss any potential memory leaks.

The `MLRef` class extends `PhantomReference` with two important pieces of information to locate the source of memory leaks (see lines 21–28). The first one is provided by the reflective AspectJ API through the `JoinPoint.StaticPart`, which keeps information about the join point, such as the location of the join point in the source. The second one allows us to keep more complete information about the calling context to be able to locate the actual source of the memory leak. To this end, we store a `Throwable` instance with the full stack trace.

The advice code in lines 8–12 shows how the weak reference to `newObject` is created by using the special predefined variable `thisJoinPointStaticPart`, and the filled stack trace with the new `Throwable` object. The `ReferenceQueue` instance passed as last argument is used by the garbage collector to place weak references that are cleared. We store all non-cleared weak references in the `refs` set. Since the advice can be invoked concurrently by several threads, we used a synchronization proxy for thread-safety.

As discussed in Section 4, the invocation of the advice happens within instrumented code. To avoid infinite recursions, the `AspectTransformer` tool is used to add the DIB support to the aspect constructor, advices, static initializer and compiler-generated methods, such as the static method `aspectOf()` (not shown).

Two threads are created by the aspect. (1) The `CleanupThread` (lines 58–67) constantly removes all cleared weak references from the `refs` set. This thread is started as a daemon thread in the aspect constructor (lines 17–18). (2) The `ShutdownThread` is responsible for the final cleanup and the actual memory leak analysis. We use a JVM `shutdown hook` to trigger the execution of this thread after all non-daemon threads have terminated. Firstly, the garbage collection is forced until the free available memory remains stable, and, in order to avoid an infinite loop in case a daemon thread continues allocating memory, we enforce the upper limit `MAX_GC` on the number of iterations (lines 37–43). Secondly, we poll the

---

11 This method is part of the sample code used to show how to detect memory leaks with IBM Rational Application Developer, which uses the LeakBot [34] technology based on the JVMPI. [http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/](http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/).

12 Array allocations can be captured by using the `joinpointfs.arrayconstructor` extension of the AspectJ weaver. We always enable this extension.

13 Even though this mechanism is rather expensive compared to our efficient calling context reification enabled by CARAJillo, here we want to show that it is possible to build simple tools using only MAJOR together with standard Java APIs.
refQueue and remove the cleared weak references from the refs set (lines 45–49). The final loop (lines 51–54) performs the actual leak analysis.

7.3. Finding memory leaks

Let’s come back to the example of Fig. 13. Firstly, we wove the LeakDetectorAspect only into application code using the standard AspectJ weaver. We invoked the method \texttt{slowlyLeakingVector(int iter, int count)} with
Fig. 15. Part of the leak analysis reported by the LeakDetectorAspect.

Arguments \texttt{iter}=1000 and \texttt{count}=10, which should normally result in 10000 allocated objects and 1000 non-reclaimed objects. The aspect, however, reported only 1 object that was not reclaimed (which corresponds to the Vector instance of Fig. 13 allocated in the static initializer), out of a total of 12 allocated objects. Of course, this result is incomplete because there is no allocation bytecode in the method.

Secondly, we wove the aspect into both the Java class library and into application classes using MAJOR. Now the aspect reported the correct information of non-reclaimed objects. Fig. 15 shows part of the analysis result. The aspect reported a total of 20157 allocated objects and 2042 non-reclaimed objects. We observe that as expected, there are 1000 occurrences of non-reclaimed String objects. In addition, 1000 non-reclaimed character arrays (\texttt{char[]} (\texttt{int})) were reported as created by \texttt{toString()}; i.e., for each entry in the vector, two objects are allocated.

In addition, we have precise information about the source of the potential memory leaks. By analyzing the stack traces in Fig. 15, we can see that the problem stems from the \texttt{slowlyLeakingVector(\ldots)} method in \texttt{LeakExample.java} at line 13. The aspect also reports 42 additional non-reclaimed objects, including objects created by the static initializer of the application class (e.g., the Vector instance itself and its internals) and other objects internally created by the JVM (e.g., created by \texttt{java.lang.Thread.<init>}, \texttt{java.lang.Shutdown.shutdown()}, etc.).

The LeakDetectorAspect is only a straw man to illustrate the benefits of aspect weaving in the Java class library. Even though all object allocations (in application code, and within the Java class library) will be captured by the aspect when the main application starts, memory leaks due to objects that are referenced only by non-daemon threads (and not by any static fields) will not be reported, because they can be reclaimed before the leak analysis. However, the aspect can be easily changed to perform the leak analysis at any moment during program execution.

### 7.4. Evaluation

For the LeakDetectorAspect, we do not measure performance but rather compare the number of allocated and non-reclaimed objects obtained when weaving the aspect with the standard AspectJ weaver and with MAJOR. We applied the aspect to the JVM98 benchmark suite and calculated some statistics.

We extended the aspect to report “application only” allocations, by filtering out those allocations that did not include any stack frame of an application method. For example, in Fig. 15 the traces including \texttt{LeakSample.main} and \texttt{LeakSample.<clinit>} are evaluated, whereas other traces that have no methods from the application class are filtered out. Here we do not perform any memory leak analysis of JVM98; a memory leak analysis of JVM98 and DaCapo can be found in [28].

Table 4 reports the number of object allocations (‘Allocations’) and non-garbage collected objects (‘Non-GC’) when running the application with the standard AspectJ weaver (setting ‘AspectJ’; unmodified Java class library) respectively with FERRARI and the AJ-UDI (setting ‘MAJOR’; woven Java class library).
Table 4: Comparison of the LeakDetectorAspect using the AspectJ weaver respectively MAJOR.

<table>
<thead>
<tr>
<th>JVM98</th>
<th>AspectJ</th>
<th>Coverage (%)</th>
<th>Non-GC Coverage (%)</th>
<th>MAJOR</th>
<th>Allocations</th>
<th>Non-GC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Allocations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>836</td>
<td>16.07</td>
<td>34</td>
<td></td>
<td>5,201</td>
<td>1,348</td>
</tr>
<tr>
<td>jess</td>
<td>7,902,648</td>
<td>99.60</td>
<td>270</td>
<td></td>
<td>7,934,455</td>
<td>3,151</td>
</tr>
<tr>
<td>db</td>
<td>155,296</td>
<td>4.84</td>
<td>9</td>
<td></td>
<td>3,210,242</td>
<td>1,284</td>
</tr>
<tr>
<td>javac</td>
<td>3,740,232</td>
<td>62.78</td>
<td>5,084</td>
<td></td>
<td>5,957,607</td>
<td>15,173</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>1,386</td>
<td>19.29</td>
<td>1290</td>
<td></td>
<td>7,186</td>
<td>2,661</td>
</tr>
<tr>
<td>mtrrt</td>
<td>6,457,760</td>
<td>97.28</td>
<td>13</td>
<td></td>
<td>6,638,029</td>
<td>1,486</td>
</tr>
</tbody>
</table>

We calculated the percentage of the values reported in the setting ‘AspectJ’ with respect to those reported in the setting ‘MAJOR’. This gives us a measure of the actual “coverage” of the aspect without weaving the Java class library (‘Coverage’ in Table 4).

We observe that the aspect is able to cover a high percentage of allocated objects for ‘jess’ and ‘mtrt’ (99.60% and 97.28%). However, the coverage for non-garbage collected objects is rather low for all benchmarks. This results in incomplete information for memory leak analysis. We also observe that the ‘db’ benchmark has the lowest allocation coverage of only 4.84%, which can be explained by the fact that most object allocations happen within the Java class library [21] and therefore cannot be intercepted without comprehensive aspect weaving.

In [9] the authors use a modified version of the Jikes RVM [27] in order to report static metrics, dynamic metrics, and performance results for the DaCapo and JVM98 benchmarks. Amongst others, they report the number of allocated and live objects. Even though we cannot make a strict comparison with our results, because of the use of different Java class library and JVMs, we obtained similar values for ‘jess’, ‘db’, ‘javac’, and ‘mtrt’ with a relative difference (by taking our measurements as reference) between 0.7% and 0.9%. For ‘compress’ and ‘mpegaudio’, the difference is higher, because our aspect also captures the allocations of arrays. Thanks to MAJOR, these results were obtained without resorting to any JVM modification. This case study also shows that the use of aspects to implement profilers and debuggers makes the approach extremely flexible because the tools can be easily modified without the burden of dealing with low-level bytecode transformations.

8. Case study 3: reproducing crashes

In this last case study, we show how MAJOR together with CARAjillo are used to implement an aspect-based debugging tool. The aspect subsumes the functionality of ReCrash [3], an existing tool that generates unit tests to reproduce program failures.

ReCrash enables 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 completion, 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 [36], 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 Java class library developers.

4. High overhead: When applied to standard benchmarks (see the evaluation in Section 8.1), ReCrash introduces high overhead.  

The authors of ReCrash reported significantly lower overhead [3]. However, instead of standard benchmarks, they used only a few applications, such as “SVNKit checkout”, which were likely I/O-bound.
Fig. 16 presents our ReCrashAspect that solves all the aforementioned limitations, thanks to CARAJillo. ReCrashAspect is similar to CCAspectFast in Fig. 11. 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 (specified by the expression 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 Fig. 10, 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, and the advice cannot know the number of callees that completed abnormally (i.e., the number of elements to pop off the shadow stack).

Crashes are detected as uncaught exceptions in the main(String[]) method. This is captured by the after() throwing(Throwable 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.

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 Fig. 17.
3. Since the instrumentation performed by MAJOR and CARAJillo 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 Java class library developers.
4. As we will show next, our ReCrashAspect outperform the original ReCrash tool on standard benchmarks.

### 8.1. Evaluation

We present our evaluation results for our ReCrashAspect and compare it with the original ReCrash tool (version 0.3). For the evaluation, we use the SPEC JVM98 and DaCapo benchmarks with the same settings and test platform as the evaluation presented in Section 6. 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 MAJOR and the original ReCrash tool leverage the java.lang.instrument API for load-time instrumentation), which primarily affects the initial program execution phase.

Fig. 16 shows a comparison of the overhead of the original ReCrash tool with the overhead cause by the ReCrashAspect. 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 with MAJOR with CARAJillo enabled. We consider two different settings, ‘App. only’ and ‘Complete’. In the former setting, the aspect is woven only into application classes, allowing a fair comparison with the original ReCrash tool. The latter setting corresponds to comprehensive aspect weaving where the aspect is also woven into the Java class library, which is not supported by the original ReCrash tool.

Table 5 shows our measurements. Regarding the ‘App. only’ 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’ is known to be particularly “exception-intensive” [17]. Hence, our aspect incurs the overhead of shadow stack cleanup in exception handlers.

In the ‘Complete’ 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

The “Twin Class Hierarchy” (TCH) [19] claims to support user-defined instrumentation of the standard Java class library. TCH replicates the full hierarchy of the instrumented Java class library in a separate package that coexists with the original one. However, this technique has the disadvantage that applications need to be instrumented to explicitly refer to a desired version of the Java class library (original or instrumented). In addition, as pointed out in [44], the use of replicated classes limits the applicability of instrumentation in the presence of native code; e.g., call-backs from native code may not reach the instrumented code. Consequently, TCH fails to transparently instrument the complete Java class library and is therefore not suited for comprehensive aspect weaving.

Unlike TCH, our approach does not use class replication techniques, but rather code duplication within the method bodies. We enable comprehensive aspect weaving independently of the presence of native methods. In addition, our approach ensures transparency for the application, which need not be aware whether the Java class library is instrumented or not.

PROSE allows runtime weaving of aspects [38], that is, aspects can be woven during the application execution. To this end, PROSE uses code hotswapping techniques, bytecode instrumentation, and an extension of the jikes RVM [27] to support code replacement while the application is running. Even though hotswapping removes the problem of bootstrapping with an instrumented Java class library (hotswapping is triggered after bootstrapping), it also imposes strong restrictions on the possible transformations (fields and methods cannot be added or removed), thus complicating the applicability in support of comprehensive aspect weaving. In addition, PROSE does not support standard AspectJ, but defines aspects and transformations as regular Java classes. PROSE does not support static join points, provides a restricted reflective interface and therefore provides limited context information within advice. In contrast, our approach aims at instrumenting the whole Java class library and efficiently reifying the calling context, while remaining compatible with standard AspectJ.

The work on MAJOR has largely inspired our work on HotWave [46], a comprehensive runtime aspect weaver. HotWave relies on hotswapping through class redefinition provided by the java.lang.instrument API, ensuring portability. HotWave relies on standard AspectJ and avoids the problem of bootstrapping with an instrumented Java class library. HotWave uses the DIB mechanism to avoid infinite recursions. In contrast to MAJOR with CARAJillo, HotWave does not support efficient calling context reification because it requires extending method signatures, which would violate current hotswapping constraints.

In [15] the authors use *J [16], a JVMPI-based tool, to gather dynamic metrics of woven programs in order to evaluate the overhead caused by AspectJ. They use annotations to keep track of the code inserted by AspectJ. Instead of resorting to a low-level tool for evaluating AspectJ, we promote using AspectJ for profiling and debugging Java programs. We currently do not consider aspects for evaluating AspectJ itself, though this could be an interesting future investigation.

The aspect-based memory leak detection approach in [31] uses aspects similar to ours but is not able to identify memory leaks if no explicit allocation is in the application code. Our approach solves exactly this issue. Other non-aspect based tools for memory leak detection, such as Cork [28] or LeakBot [34], are able to detect leaks within the Java class library, but rely on customized JVMs and on native code, thus trading portability for performance.

In Smalltalk, the call stack is directly accessible due to the reflective nature of the language. AspectS [25], 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 [42] and JBoss AOP [26] using the Throwable API, resulting in high overhead.

Steamloom [22,10] provides 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 [27] supporting efficient aspect execution and runtime aspect weaving. This approach also trades portability for performance, since the aspect support is integrated in the JVM. In contrast, our approach supports comprehensive and portable aspect weaving, while providing complete calling context information with moderate overhead.

Our approach to calling context reification uses before and after pointcuts to maintain a shadow stack. While it is possible to reify the calling context using the cflow pointcut,15 such an approach would introduce high overhead [4,24]. The AspectJ and abc [5] compilers rely on thread-local counters 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 everywhere, thus considerably reducing the access to the thread-local variable. More efficient implementations of cflow exploit direct access to JVM internals and can be integrated into the just-in-time compiler [11]; however, they require a modified JVM.

---

15 The control flow of a join point, captured by cflow, defined the flow of the program instructions that occur as a result of the invocation of the join point [32], which relates to the call stack.
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 [20] 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) [2], which is commonly used for profiling and program analysis. Existing approaches that create accurate CCTs [39,2] suffer from considerable overhead. Approaches based on sampling and stack-walking [49] help reduce the overhead, but at the expense of a loss of accuracy. 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. We successfully applied the approach described in this article to efficiently create accurate and complete CCTs, and combined it with techniques to parallelize application execution with CCT generation on multicores [6].

10. Conclusion

In this article we presented techniques and tools that enable comprehensive aspect weaving. Our approach relies on a generic framework that enhances user-defined instrumentations with complete method coverage. We adapted the popular AspectJ weaver to our framework without changing any AspectJ sources. We succeeded in removing a serious limitation from AspectJ, its inability to weave aspects into the Java class library. Our approach makes aspect-based profiling and debugging techniques practical, since it ensures complete method coverage. Moreover, it makes AOP available to the developers of the Java class library.

We also 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 and debugging tools that require accurate calling context information as well as complete method coverage.

We validated and evaluated our framework with existing and new profiling and debugging aspects, and showed the efficiency of our approach. First, we wove existing profiling aspects with our framework in order to produce complete profiles representing overall program execution. This case study confirms that our approach is largely compatible with existing aspect code. Second, we wrote a simple aspect for memory leak detection and showed with an example that comprehensive aspect weaving is indispensable for aspects that perform dynamic program analysis. Third, we recasted an existing testing tool, which was based on low-level bytecode instrumentation techniques, as an aspect. In comparison with the original tool, the aspect is compactly represented in a few lines of source code, offers improved functionality, eases the integration of extensions, benefits from complete method coverage, and causes less overhead.

Our approach shares the limitations with any design based on bytecode instrumentation, such as the impossibility of instrumenting native code. We however attenuate this limitation by using native method prefixing to correctly represent native method calls in calling context representations. Another limitation concerns potential code bloat due to the different composed transformations, each of them contributing to the overall code bloat.

In summary, our original use of AOP as a high-level representation of instrumentations enables the rapid development of sophisticated and efficient profiling and debugging tools.

Acknowledgement

The work presented in this article has been supported by the Swiss National Science Foundation as part of the project FERRARI (project number 200021-118016/1).

References
