Università
della
Svizzera
italiana

**Software
Institute**

SOFTWARE & DATA ENGINEERING MASTER THESIS

Alessio Giovagnini

Advisor: Dr. Andrea Rosà, Co-Advisor: Dr. Lubomir Bulej

FROM ASM TO CLASS-FILE API: THE CASE OF DISL

Abstract

The Class-File API is a recent addition to the Java Class Library, introduced as a preview feature in JDK 22. The API provides support for parsing, generating, and transforming Java class files, which, before the introduction of the API, was possible only via third-party libraries such as ASM, BCEL, or Javassist. Several dynamic-analysis frameworks (e.g., DiSL) currently rely on these third-party libraries to manipulate bytecode, and even the JDK uses these libraries (in particular, ASM) to implement tools such as jar and jlink, and to support the implementation of lambda expressions at runtime. However, the dependency of dynamic-analysis frameworks and of the JDK on these libraries is increasingly becoming problematic, particularly after the adoption of the six-month release cadence of the JDK, as the class-file format is evolving more quickly than in the past. On the one hand, relying on the willingness of third-party developers to update and test these libraries is not ideal, particularly when the libraries are used by the JDK. On the other hand, these libraries can be updated only with a considerable delay after the release of a new class-file format, which undermines the uptake of new class-file features across the Java ecosystem and dynamic-analysis frameworks. The Class-File API aims at removing this dependency, as bytecode manipulation can now be done with the standard Class-File API that evolves together with the class-file format. This project recasts the dynamic-analysis framework DiSL to use the new Class-File API instead of ASM. In this document, we will illustrate the main challenges we faced, the solutions we found, as well as useful insight into the Class-File API.

Contents

List of Figures

List of Tables

Acknowledgments

I sincerely thank my advisors, Andrea Rosà and Lubomir Bulej, for helping me along the development of this thesis. I would also like to thank all the USI staff and my family for the support they gave me. Finally, I would like to thank my friend Jacob Salvi for letting me use his profiler for testing.

1 Introduction

Bytecode instrumentation is the process of editing the bytecode of a Java class file. This process is performed to add specific functionality to selected sections of a program, even without access to the source code. The instrumentation usually aims to maintain the behavior of the program as close to the original as possible in both functionality and duration of execution. There are many metrics and information that are not made explicit by the environment or the runtime, but can be obtained by instrumenting the program. Some examples include monitoring object creation or following the sequence of method calls. Many dynamic analysis tools for the *Java Virtual Machine* (JVM) use bytecode instrumentation to implement their functionality. DiSL [1] is one of such tools and the subject of this thesis. DiSL is short for *domain-specific language for instrumentation* and is a framework to perform bytecode instrumentation using its designed language, Java, with special annotations. Since it instruments bytecode, it is not limited to applications compiled with Java, but it can perform dynamic analysis on any program that can run on the JVM. DiSL uses principles of aspect-oriented programming (AOP), but unlike similar competitors like AspectJ [2, 3, 4], the DiSL weaver allows for full-code coverage without limitation. This is thanks to the open join point model, which provides the ability to mark any bytecode region defined by the user. Other notable features of DiSL are the access to both static and dynamic context information efficiently, and generally producing significantly less overhead during the analysis than the competitors.

DiSL is written mostly in Java, and internally it relies on the ASM [5] library. ASM allows users to read, analyze, and edit instructions of compiled Java classes. The process of bytecode manipulation is eased by the high-level API offered; everything is wrapped in a higher level of abstraction, and the user does not have to interact with binary components.
The ASM library is not officially supported by Oracle; it is completely maintained by third-party developers. Nonetheless, ASM comes bundled with the JVM. More specifically, the library is internally used to implement tools like *jar* [6] and *jlink* [7]; it is also essential to support the implementation of lambda expressions at run time. Relying on a third-party library for essential tasks such as bytecode manipulation is not ideal. The class-file format is in constant evolution, and in recent years, it has been evolving quickly due to the 6-month release cadence of the JDK. This causes significant delays to certain features related to the class-file format, since at the release of the JDK version N, the changes won't be implemented in ASM yet, and the JDK will have to wait for version N+1 to safely include the newer features.
These delays also negatively affect DiSL. With every JDK update, there will be a period where some features might be broken until the ASM library is updated.
Since JDK 22, there has been a proposal [8] by OpenJDK to implement a new *Class-File API*; the goals are to provide an API for processing class files that tracks the class file format defined by the JVM and to enable all JDK components to migrate to this new API and eventually remove all dependencies from ASM. The Class-File is intended to perform bytecode generation and manipulation, so in many regards it is similar to ASM; however, since ASM was created, Java has introduced many new language features, which the Class-File takes advantage of to provide a more flexible and less error-prone API. The new Class-File API was included as a preview in JDK 22. It has been finalized and fully included in JDK 24.

1.1 Contributions

The main contribution of this thesis is the recasting of DiSL to internally use the Class-File API to perform instrumentation and the removal of all ASM dependencies.
We provide an analysis comparing the performance of our version of DiSL with the Class-File and the original DiSL, which uses ASM. The analysis is performed using already established profilers such as P^3 [9] on different well-respected benchmarks included in Renaissance [10]. Additionally, we provide many useful observations on the Class-File API, we compare its features with ASM, and we explain in detail the implementations we adopted during the recast.

1.2 Challenges

Replacing a library in a program can pose various challenges. The compatibility between the old and the new library is one of the largest sources of problems. While refactoring DiSL, we faced many issues caused by subtle differences in the APIs and architectures. The other major problem and most difficult challenge was testing the correctness of the refactored application. ASM and the Class-File API are not compatible, and it is not possible to switch from one data structure to the other automatically. Most of the DiSL classes that utilize ASM depend on other classes that use ASM as well. This chain of dependency meant that the majority of classes couldn't be tested until all of DiSL was recast. While replacing ASM with DiSL, small errors were introduced, mostly caused by misinterpretation of the API similarities. The accumulation of all the bugs resulted in a huge effort in debugging. Static helper functions and some independent modules were unit-tested during the development and generally included far fewer bugs.

1.3 Outline

This document will present our findings and work as follows. Section 2 will briefly explore DiSl competitors and alternative bytecode manipulation libraries. In Section 3, we provide an overview of DiSl, ASM, and the Class-File. In Section 4, we will first explain our design goal, and then we will illustrate in detail the differences between ASM and the Class-File API. We will also explain the most notable problems we encountered and the solution we developed. In Section 5, we will give an analysis of the timing performance of DiSL in both versions. Finally, in Section 6, we will discuss some limitations of our implementation and potential future work.

2 State-of-the-art

In this section, we will briefly present other bytecode libraries, instrumentation frameworks, and studies. To the best of our knowledge, no other study has yet been performed directly on the Class-File API.

2.1 Bytecode Manipulation Libraries

2.1.1 ASM

ASM [11, 12] is currently the most popular bytecode manipulation library in Java. Some of the most notable uses are in the *OpenJDK* [13], *Nashorn compiler* [14], *Groovy compiler* [15] and the *Kotlin compiler* [16]. At the moment, ASM is considered the industry standard as it includes all features for bytecode manipulation and analysis. It requires a good knowledge of bytecode instructions and the class-file format.

2.1.2 BCEL

The *Byte Code Engineering Library* [17] is maintained by Apache Commons; it requires a good proficiency with the Java instruction set. Additionally, it includes a bytecode verifier for debugging faulty programs

2.1.3 Javaassist

Javaassist [18, 19] provides a very high-level API; the user needs very little knowledge of the specification of the Java bytecode. The base Java reflection API allows for introspection of data in a Java program, but has a limited capability in altering program behavior. Javaassist allows structural reflection, which is the ability to alter data structures of the program. The operation is applied before the program is loaded into the runtime system.

2.1.4 Byte Buddy

Byte Buddy [20] is a Java library for code generation and manipulation. It can create and manipulate class files during the run-time of the application. Additionally, it offers a convenient high-level API while using ASM underneath.

2.1.5 Cglib

Cglib [21] is short for *Code generation library* and is a bytecode instrumentation library. It is used by some popular Java frameworks such as *Spring* [22] and *Hibernate* [23]. ASM is listed as a dependency for this library.

2.1.6 Serp

Serp [24] is a framework for manipulating Java bytecode. It offers a high-level API and also direct access to the low-level data of the Class-Cile and constant pool.

2.2 Instrumentation Tools and Frameworks

2.2.1 AspectJ

AspectJ [2, 3, 4] is an *aspect-oriented* [25, 26] extension for the Java language. It provides support for two kinds of crosscutting implementation. One is *dynamic crosscutting*, which

allows developers to define additional implementations at some well-defined point that will be run during the execution of the program. The other is *static crosscutting*, which allows instead to define new operations on existing static types. The downside of AspectJ is that the join point model weaver does not allow for full bytecode coverage, which limits the usability of this extension in the context of instrumentation.

2.2.2 BISM

BISM [27, 28] is short for *Bytecode-Level Instrumentation for Software Monitoring*; it is a high-level bytecode instrumentation tool that follows the aspect-oriented programming paradigm. It uses ASM underneath, and it provides the following features. Access to the complete static context and also to dynamic context objects. BISM can generate the control flow graph of a target method and provides specific control-flow related join-points to capture conditional branches. It can run as a standalone application and perform static instrumentation, and also run with an agent to perform instrumentation before the linking phase. In the second mode, BISM can also perform instrumentation on additional classes.

2.2.3 JNIF

JNIF [29] is for *Java native instrumentation framework*, and it is a bytecode rewriting framework implemented in C++. It can analyze and manipulate Java class files with solid performance as it is implemented in native code.

2.3 Performance Comparison

A study [30] on overhead introduced by instrumentation frameworks has been performed comparing the following technologies: AspectJ, ByteBuddy, DiSL, Javassist, and pure source code instrumentation. The experiment performed revealed that ByteBuddy, DiSL, Javassist, and source instrumentation achieve low overhead.

2.4 Java Library Migration

Library migration is a frequent phenomenon in software development; different studies [31, 32] have been performed on the subject to find the most common reasons for migration and the prevalent domains of the libraries.

3 Background

In this section, we will provide some more in-depth information on the three main topics of the thesis: DiSL, ASM, and the Class-File API. For the two libraries, we will provide some code examples on the basic functionality of the API.

3.1 DiSL

We already briefly introduced DiSL at the beginning. Now we will give more in-depth information on the different DiSL components. We will start by introducing DiSL Snippets.

3.1.1 Snippets

Snippets are written by the user in Java; they are basically a code template that the user can use to access method information.

The annotation on a method in combination with the marker specifies the location in which the snippet will be inserted; in some cases, additional arguments can also be passed depending on the context. In Figure 1, the annotation dictates that the snippet will be inserted before each bytecode with opcode *getfield*, the *@Before* indicates the position, the marker *BytecodeMarker* means that we are dividing the method at the bytecode level, and the argument *getfield* is used to filter which bytecode we desire.
Another example is shown if Figure 2, where the marker is a *BodyMarker*, it indicate that we are considering the instrumentation on the entire body of the method, *@Before* will insert the snippet before any instruction in the method, while *@After*, will insert the snippet at the end of the method, just before returning.

Annotations can be declared on variables. For example, *@ThreadLocal*, shown in Figure 1, is a variable that will be instrumented as an instance variable inside the *Thread* class in *java.lang* package. The thread-local variable can be accessed by different snippets to share information. The annotation *@SyntheticLocal*, shown in Figure 2, is going to be instrumented as a local variable in the method and can be shared by snippets inserted in the same method. In this case the snippet with *@Before* and *@After* will share the synthetic local variable.

Inside the snippet, it is possible to access multiple kinds of static contexts, which provide information depending on the region in which they are woven. For example, in Figure 3, the context can access the name of the field being accessed.

DiSL also provides a single dynamic context to access information that is available only at runtime. The snippet shown in Figure 4 makes use of the dynamic context to access the argument input of a function and its output, which is going to be available on the stack.

Additionally, DiSL allows users to define their own custom context. Figure 5 shows an example. In a custom context, the user can access the original instructions of the method that is instrumented; the instructions are represented with ASM objects. In the example, we access the index of the instruction to get the dimensions of the multi-array. Figure 6 shows how the custom context was used.

```
public class Instrumentation {

    @ThreadLocal
    static long threadLocalVariable;

    @Before(marker=BytecodeMarker.class,
            args="getfield")
    static void readInstanceFields() {
        threadLocalVariable++;
    }
}
```

Figure 1. Example of a DiSL snippet

```
public class Instrumentation {

      @SyntheticLocal
      static long start;

      @Before(marker=BodyMarker.class)
      static void onMethodEntry() {
            start = System.nanoTime();
      }

      @After(marker=BodyMarker.class)
      static void onMethodExit() {
            System.out.println(
                "Method duration " + (System.nanoTime() - start));
      }
}
```

Figure 2. A snippet using a synthetic local variable

```
public class Instrumentation {

    @Before(marker=BytecodeMarker.class,
            args="getstatic"
    ) static void readStaticField(FieldAccessStaticContext context) {
        System.out.println(context.getName());
    }

}
```

Figure 3. A snippet that uses a context to access method information

```java
public class Instrumentation {
    // method foo --> public int[] foo(int var)
    @AfterReturning(marker = BodyMarker.class, scope="MyClass.foo")
    static void afterMethodFoo(DynamicContext dc) {
        // Returns the value of a particular method argument.
        int input = dc.getMethodArgumentValue(0, int.class);
        // Returns the value of a particular item on the JVM operand stack.
        int[] out = (int[]) dc.getStackValue(0, Object.class);
    }

}
```

Figure 4. A Snippet that accesses a Dynamic context

```java
public class CustomContext extends InstructionStaticContext{
    public int getDimension() {
        List<CodeElement> list = staticContextData.getMethodModel()
            .instructions();
        int index = this.getIndex(); // index of the instruction
        CodeElement node = list.get(index);
        switch(node){
            case NewMultiArrayInstruction n -> {
                return n.dimensions();
            }
            default -> {}
        }
        return 0;
    }
}
```

Figure 5. Example of a Custom Context

```java
public class Instrumentation {
    @Before(marker=BytecodeMarker.class, args="multianewarray",
        scope="MainThread.*")
    public static void multiArrayAllocation(DynamicContext dc,
        CustomContext cc) {
        String desc = cc.getType();
        int dims = cc.getDimension(); // dimensions of the multy array
        // ...
    }

}
```

Figure 6. Use of the Custom Context

3.1.2 Instrumentation API

This API includes all of the annotations, markers, and contexts that users use in the snippets. Annotations and markers are the components that dictate the locations where the snippets are inserted. Annotations are simple interfaces, while markers are classes where each kind of marker implements a specialized function *mark* to calculate the location within the methods; they are both used in combination by the weaver for instrumentation. Static contexts implement all the methods to obtain the static information, while the dynamic context is simply an interface since it is the weaver that inserts the functionality to retrieve dynamic information.

3.1.3 Instrumentation Library

This library provides a very simple API to configure the DiSL instance and start the instrumentation of the bytecode.

3.1.4 DiSL Weaver

The DiSL weaver is the component that instruments the snippets in the methods, and is probably the most complex part of DiSL, which took the longest to recast to the Class-File API

3.1.5 DiSL Server

The DiSL server is written in Java and is the component that uses the instrumentation library. In the server, the following steps are performed to instrument a class: first, the DiSL snippets written by the user are prepared, and then the server waits for an instrumentation request. When a request finally comes, the server will check for matching classes and methods. All the methods that have a match are then instrumented; the process is done by the DiSL weaver, which will insert the snippets in the desired locations. When the class is instrumented, it is sent back to be loaded in the JVM.

3.1.6 Java Agent

To perform instrumentation at runtime, a Java agent is attached to the JVM at startup. When the target application is running, the agent intercepts all classes before they are loaded into the JVM and sends them to the DiSL server for instrumentation. After being instrumented, the class is sent back and is loaded into the JVM. The advantage of instrumenting at run-time is that every class used in the application can be instrumented, even classes in the Java core library. If the instrumentation was statically executed before starting the application, only the classes included in the application itself could be instrumented. The agent is written in *C++*; we did not make any changes to the DISL agent as it is completely independent of the instrumentation process performed by ASM.

3.2 ASM

The ASM library provides two primary systems for manipulating bytecode. The core package *org.objectweb.asm* uses a visitor pattern to transform classes and methods. The other package *org.objectweb.asm.tree* instead uses a tree representation where every component can be accessed and edited in any order. The two API are comparable to the Stream API for XML processing and the Document Object Model (DOM) XML processing. The core package is faster and requires less memory, but it can be limiting in certain situations since only one element at a time is available for manipulation. Additionally ASM provides an analysis package

org.objectweb.asm.tree.analysis with semantic analyzers, interpreters, and verifiers.

The example shown in Figure 7 is a class visitor that simply prints the name of the field and methods. The visitors can also edit the different class elements when they are being visited. Another approach for parsing classes is using the tree API, as shown in Figure 8.
It is possible to access all the elements using this API. In certain situations, the tree API might be more convenient, even with extra cost in performance, since it allows more control over the reading and writing elements. In DiSL, the tree API is the most used method to edit the bytecode.

```
public class MyClassVisitor extends ClassVisitor {

    public MyClassVisitor() {
        super(Opcodes.ASM9);
    }

    public FieldVisitor visitField(int access, String name,
                                   String desc, String signature,
                                   Object value)
    {
      System.out.println("Visited field: " + name);
      return super.visitField(access, name, desc, signature, value);
    }

    public MethodVisitor visitMethod(int access, String name,
                                     String desc, String signature,
                                     String[] exceptions)
    {
        System.out.println("Visited method: " + name);
        return super.visitMethod(access, name, desc, signature, exceptions);
    }
}
```

Figure 7. A simple ASM visitor

```java
final ClassNode result = new ClassNode(Opcodes.ASM9);
new ClassReader(bytes).accept(result, ClassReader.EXPAND_FRAMES);

for (FieldNode fieldNode: result.fields) {
    // ...
}

for (MethodNode methodNode: result.methods) {
    for (AbstractInsnNode abstractInsnNode: methodNode.instructions) {
        if (abstractInsnNode instanceof InsnNode) {
            // ...
        }
    }
}
```

Figure 8. Example of parsing with ASM tree API

3.3 Class-File API

The Class-File API presents some similarities to ASM; classes have a tree structure that the user can navigate. One essential difference is that each entity is immutable; the user can't directly create a new object, as all the constructors are private. Instead, the Class-File exposes its API only through sealed interfaces. To perform transformations, three abstractions are provided, *elements*, *builders* and *transformers*. Elements represent a component of the class that can range from a portion of code, of a class, or of a method. All elements are immutable. A builder is a consumer where corresponding elements can be passed; for example, a *CodeBuilder* will accept *CodeElement* objects to generate the code of a method. The transformer is a function that accepts an element and a builder; the user can then define inside the behavior to transform or generate the class, method, or code.
The Figure 9 shows how to access the elements in the tree. First, the bytes of a class are parsed to obtain a *ClassModel*, then we can access all the components of the class.

```java
ClassModel cm = ClassFile.of().parse(bytes);
for (ClassElement ce: cm) {
    if (ce instanceof MethodModel mm) {
        for (MethodElement me: mm) {
            if (me instanceof CodeModel xm) {
                for (CodeElement e: xm) {
                    if (e instanceof Instruction i) {
                        // ...
                    }
                }
            }
        }
    }
}
```

Figure 9. Parsing a class with the Class-File API

Figure 10 shows a class, with a main method and a constructor being built. The function *build*

takes a constant descriptor and a consumer function. The input argument of the function is the builder; by invoking different methods on the builder, it is possible to add components to the class, like methods and fields. Constructing the method works the same; a consumer with a method builder is used. To add the code, a consumer with a code builder is passed, and all the instructions are passed to the builder. There are different builders for different components of the class; each offers methods that specialize in its respective role. The code presented appears to be quite convoluted. To keep the code cleaner and more readable, each lambda could be declared as a separate function as shown in Figure 11.

```
byte[] bytes = ClassFile.of().build(CD_Hello,
        classBuilder -> classBuilder.withFlags(ClassFile.ACC_PUBLIC)
            .withMethod(
                ConstantDescs.INIT_NAME,
                ConstantDescs.MTD_void,
                ClassFile.ACC_PUBLIC,
                methodBuilder -> methodBuilder.withCode(
                    codeBuilder ->
                        codeBuilder.aload(0)
                                   .invokespecial(
                                        ConstantDescs.CD_Object,
                                        ConstantDescs.INIT_NAME,
                                        ConstantDescs.MTD_void)
                                   .return_()))
            .withMethod(
                "main",
                MTD_void_StringArray,
                ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC,
                methodBuilder -> methodBuilder.withCode(
                    codeBuilder ->
                        codeBuilder.getstatic(
                                        CD_System,
                                        "out",
                                        CD_PrintStream)
                                   .ldc("Hello World")
                                   .invokevirtual(
                                        CD_PrintStream,
                                        "println",
                                        MTD_void_String)
                                   .return_())));
```

Figure 10. Building a class with the Class-File API

```
Consumer<MethodBuilder> mb1 = (methodBuilder -> methodBuilder.withCode(
                codeBuilder ->
                        codeBuilder.aload(0)
                                .invokespecial(
                                        ConstantDescs.CD_Object,
                                        ConstantDescs.INIT_NAME,
                                        ConstantDescs.MTD_void)
                                .return_()));

Consumer<MethodBuilder> mb2 = (methodBuilder -> methodBuilder.withCode(
        codeBuilder ->
                codeBuilder.getstatic(
                                CD_System,
                                "out",
                                CD_PrintStream)
                        .ldc("Hello World")
                        .invokevirtual(
                                CD_PrintStream,
                                "println",
                                MTD_void_String)
                        .return_()));

Consumer<ClassBuilder> cb = (classBuilder ->
    classBuilder.withFlags(ClassFile.ACC_PUBLIC)
        .withMethod(
                ConstantDescs.INIT_NAME,
                ConstantDescs.MTD_void,
                ClassFile.ACC_PUBLIC,
                mb1)
        .withMethod(
                "main",
                MTD_void_StringArray,
                ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC,
                mb2));

byte[] bytes = ClassFile.of().build(CD_Hello, cb);
```

Figure 11. Class-File transformers separated

If, instead of creating a new class, we want to modify an existing one, we can use transformers, as shown in Figure 12. Transformers are functions that accept two inputs, a builder and an element. They can be treated as a *flatMap* operation, where the elements are passed one by one, and the function defined by the user can replace the element, drop it, or pass it unchanged. There are transformers for every level of the Class-File, a *MethodTransform* accepts a *MethodBuilder* and a *MethodModel*, while a *CodeTransform* accepts a *CodeBuilder* and a *CodeElement*, an example is shown in Figure 13.
The disadvantage of a transform is that during the transformation, we do not have access to the previous or next element. In some cases, where the context is needed to decide the operation, it might be easier to first parse and obtain the list of all elements needed, then manually modify the list, and finally rebuild the original class with some of the elements changed.
In DiSL, we used both methods with builders and transformers depending on the situation.

```
ClassTransform classTransform = (builder, element) -> {
    if (element instanceof MethodModel method) {
        // do something with the method
    } else {
        builder.with(element); // pass the element unchanged
    }
};
byte[] newBytes = ClassFile.of()
    .transformClass(oldClassModel, classTransform);
```

Figure 12. Transforming a class with the Class-File API

```
CodeTransform codeTransform = (codeBuilder, codeElement) -> {
    if (codeElement instanceof OperatorInstruction oi
        && oi.opcode == Opcode.IADD) {
        codeBuilder.imul(); // replace iadd with imul
    } else {
        codeBuilder.with(codeElement);
    }
};
```

Figure 13. A Class-File code transformer

3.4 Glossary

This section briefly explains some of the most commonly used terms in this thesis and describes the ASM and Class-File classes mentioned.

JVM Java Virtual Machine.

Hot Spot Most used JVM implementation.

JIT Just in time compiler.

Jar General-purpose archiving and compression command line tool. An extension of a Java archive containing Java class files.

Jlink Command line tool to link a set of modules, along with their transitive dependence, to create a custom runtime image.

C2 *Server Compiler* used by the JVM for optimizing *Hot* methods.

3.4.1 ASM Classes

AbstractInsnNode A node that represents a bytecode instruction.

ClassNode A node that represents a class.

FieldNode A node that represents a field.

MethodNode A node that represents a method.

InsnList A doubly linked list of AbstractInsnNode objects

InsnNode A node that represents a zero-operand instruction.

IntInsnNode A node that represents an instruction with a single int operand.

JumpInsnNode A node that represents a jump instruction.

VarInsnNode A node that represents a local variable instruction. A local variable instruction is an instruction that loads or stores the value of a local variable.

Type A Java field or method type.

LabelNode An AbstractInsnNode that encapsulates a Label.

Label A position in the bytecode of a method.

ClassVisitor A visitor to visit a Java class.

FieldVisitor A visitor to visit a Java field.

MethodVisitor A visitor to visit a Java method.

3.4.2 Class-File API Classes

ClassModel Models a class file.

MethodModel Models a method.

CodeModel Models the body of a method.

FieldModel Models a field.

ClassBuilder A builder for a class file.

MethodBuilder A builder for methods.

CodeBuilder A builder for Code attributes (method bodies).

FieldBuilder A builder for fields.

ClassFileElement Marker interface for structures with special capabilities in the class file format.

ClassElement Marker interface for a member element of a ClassModel.

MethodElement Marker interface for a member element of a MethodModel.

CodeElement Marker interface for a member element of a CodeModel.

FieldElement Marker interface for a member element of a FieldModel.

ClassTransform A transformation on streams of ClassElement.

MethodTransform A transformation on streams of MethodElement.

CodeTransform A transformation on streams of CodeElement.

FieldTransform A transformation on streams of FieldElement.

LabelTarget A pseudo-instruction which indicates that the specified label corresponds to the current position in the Code attribute.

Label A marker for a position within the instructions of a method body.

Attribute<T> Models an attribute in the class file format.

CustomAttribute Models a user-defined attribute in a class file.

Instruction Models an executable instruction in the code array of the Code attribute of a method. The order of instructions in a CodeModel is significant.

StackInstruction Models a stack manipulation instruction in the code array of a Code attribute.

3.4.3 Java Classes

ClassDesc A nominal descriptor for a Class constant.

MethodTypeDesc A nominal descriptor for a MethodType constant.

URLClassLoader This class loader is used to load classes and resources from a search path of URLs referring to both JAR files and directories.

Iterable Interface, if implemented, it allows an object to be the target of the enhanced for statement.

Optional<T> A container object which may or may not contain a non-null value.

4 DiSL recast

In this section, we will first explain our design goal for the recast, and we will illustrate some of the differences between the Class-File API and ASM. Then we will explain all the major problems we faced during the recast with the implementation of the solution used.
Additionally, we provide information on our testing and debugging process.
The code of our recast version of DiSL can be found in this repository: https://github.com/usi-dag/disl-classfile-api.

4.1 Design Goals

4.1.1 Functionality

Features and Functionality should remain the same in the Class-File version of DiSL. While recasting, we should not change the DiSL API that users interact with in the snippets. By removing ASM from DiSL, the users won't be able to access ASM instructions in custom contexts, but it will be possible to access Class-File instructions instead. This change will make some profilers incompatible with our version without a recast, but we decided that the change was necessary. Section 4.3 will provide more details.

4.1.2 Instrumentation

We decided that the instrumented bytecode produced by the Class-File version of DiSL shouldn't be identical to the one produced by the ASM version of DiSL, but should be semantically equivalent.

4.1.3 Performance

We decided that the recast version of DiSL should have a performance similar to the original DiSL to be considered a valid replacement; the analysis on performance is provided in Section 5.

4.2 Class-File and ASM Differences and Similarities

We will first introduce some of the major differences and similarities between the two API and provide some examples.

4.2.1 Tree Structure

The comparison between the ASM tree API and the Class-File shows many similarities. In Figures 14 15, the internal representation of each library for the main structure of a Java class file is shown.
One difference is in the data access; in ASM, many components can be accessed from the fields of the object, while the Class-File API has private fields and uses methods to return the data in an immutable form. Another difference is how instructions are obtained; in ASM, the list of instructions is obtained from the *MethodNode* 3.4.1, while the Class-File stores the list of instructions in the *CodeModel* 3.4.2, which is returned as an *Optional* 3.4.3 from the *MethodModel* 3.4.2.

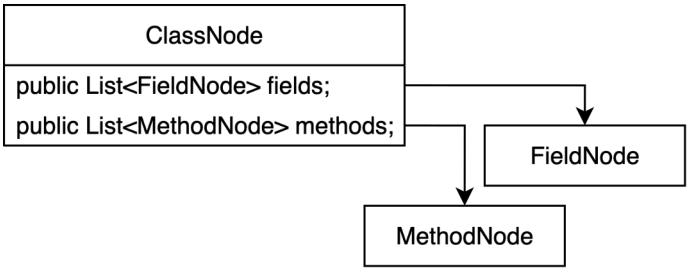


Figure 14. ASM classes

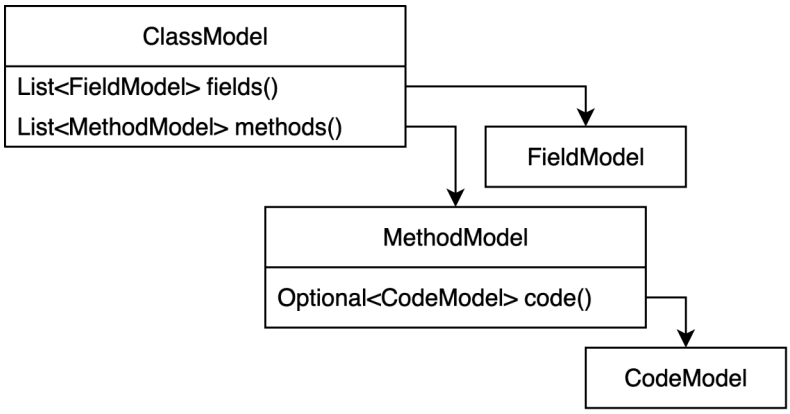


Figure 15. Class-File classes

4.2.2 Instruction List

In ASM, we can obtain the list of instructions from a *MethodNode* as an *InsnList* 3.4.1. This data structure is a double-linked list containing *AbstractInsnNode* 3.4.1 objects; the linked list can also be accessed by index and can be iterated as it implements *Iterable* 3.4.3.
The Class-File API uses instead an immutable *java.util.List* of *CodeElement* 3.4.2, this structure is missing some of the functionality of the ASM *InsnList*, such as methods to directly access the next and previous instruction. This functionality is widely used in the original DiSL code; we adapted it using helper functions and by passing the list of instructions around.

4.2.3 Mutability

As already stated previously, all elements in the Class-File API are immutable, which poses problems since in the weaver we need the ability to add, remove, and edit instructions to insert snippets of code inside methods. A simple example is when the snippet includes local variable slots that would conflict with the method slots. In ASM, this process would simply require scanning the list of instructions and updating all slot accesses. With the Class-File, we need instead to transfer all instructions in a new mutable list, since the original list obtained from the method is also immutable. Then all the instructions that need an update will be discarded, and a new one with the correct slots will be inserted to replace the original. The procedure is not as efficient as ASM; alternatives could be implemented, such as using chained transformers. We tried to use Class-File transformers as much as possible while refactoring DiSL, but in some cases we had to adopt other implementations in the weaver to avoid complex refactoring. Instead of using Class-File transformers, we first transferred all the instructions on mutable lists, and then we performed the instrumentation by directly editing the list of instructions.

4.2.4 Sealed Interfaces and Custom Elements

All elements in the Class-File API are sealed interfaces with one exception: the class *Custom-Attribute* 3.4.2 exists to let users define their own Class-File attributes. The class implements all the following interfaces: *Attribute<T> 3.4.2, ClassElement 3.4.2, ClassFileElement 3.4.2, CodeElement, FieldElement, and MethodElement*. This allows mixing custom elements and normal elements in data structures and method parameters.

4.2.5 Instructions

Both ASM and the Class-File API represent bytecode instructions using a mix of pseudo-instructions and real instructions.
The Figure 16 compares only real instructions that represent a JVM Opcode. The two element at the top, *Instruction* 3.4.2 and *AbstractInsnNode* 3.4.1 are respectively the super-interface and super-class of all the elements below. Half the instructions of ASM have a direct equivalent to the Class-File. If we exclude the discontinued instructions, *JumpInsnNode* 3.4.1 also has a direct mapping, while *VarInsnNode* 3.4.1 is subdivided in the Class-File to differentiate storing and loading. ASM groups all zero-operand instructions in the same class *Insnode* 3.4.1, while in the Class-File, they are all subdivided into different interfaces. This division allows for better parsing using pattern matching in switch cases, while using ASM requires pattern matching on the Opcode.
Another advantage of subdividing the instructions into more specialized cases is in function parameters. For example, with the Class-File, if we want a class that only accepts instructions relevant to stack operation, we can declare the parameter as *StackInstruction* 3.4.2, while with ASM, we would require additional assertions, before or inside the function.

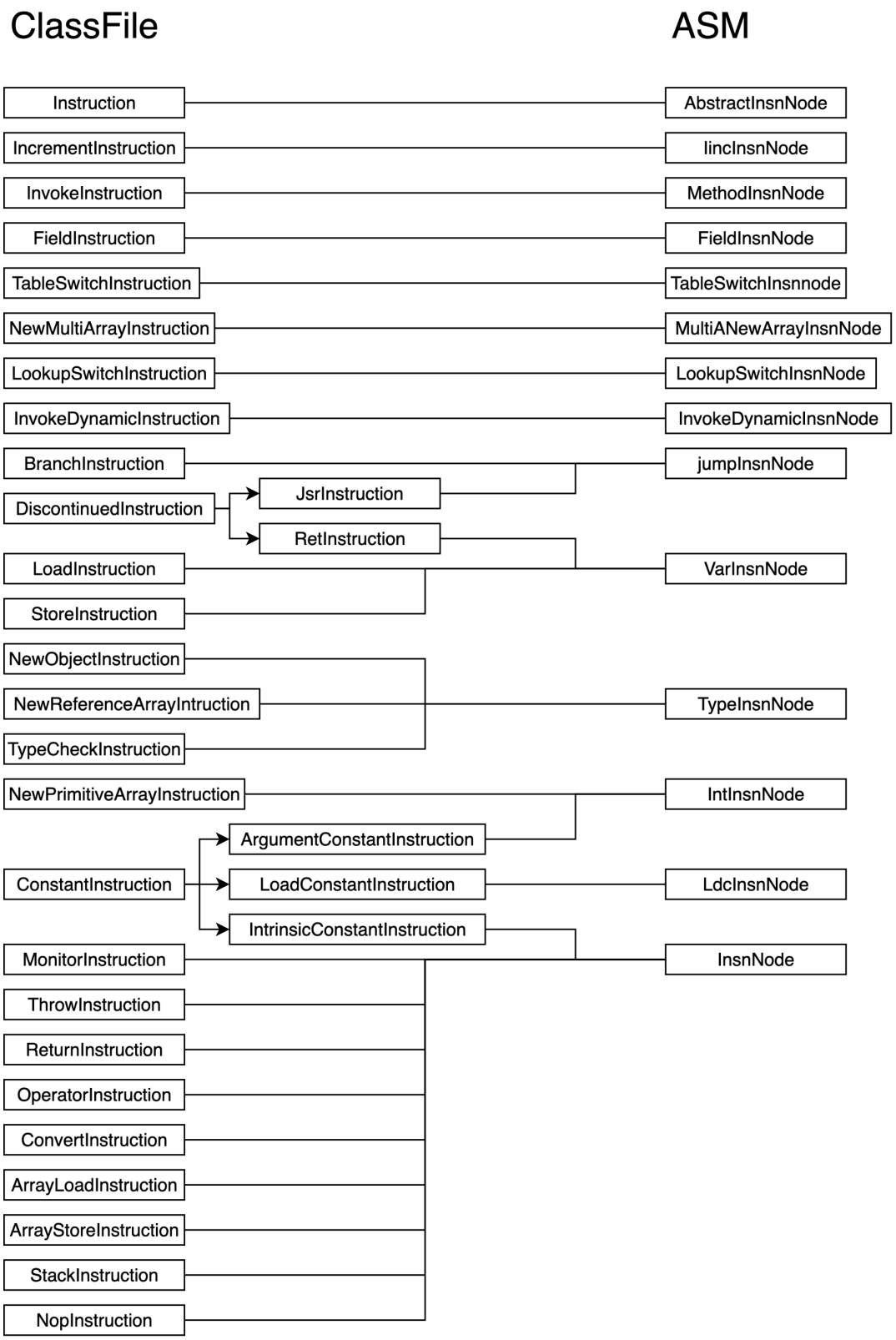


Figure 16. Comparison between real ASM and Class-File instructions

4.2.6 Classes and Methods Type Representation

The ASM *Type* 3.4.1 class is used to represent both a field and method type. The class includes methods that only work with one of the two, meaning that it falls upon the user to use the correct functions. This ambiguity can also be confusing when reading code if it is not properly documented. The Class-File adopts a different approach and instead uses two different classes. *ClassDesc* 3.4.3 is used to represent a nominal descriptor for a class constant, *MethodTypeDesc* 3.4.3 is used to represent a method constant. The separation guarantees that the user will never call the *wrong* function, and it will also make explicit which data type is being used. The parameters and return type of a method represented by a *MethodTypeDesc* will also be returned as *ClassDesc* objects.

4.2.7 ASM Traversal, Visitor

The original version of DiSL made use of both the tree API for traversal and several visitor classes for transformations. DiSL uses *visitors* three times; a class visitor is used for inserting a thread local variables defined in snippets inside the class *java.lang.Thread*. The second is a method visitor, it is implemented inside the DiSL marker *AfterInitBodyMarker* to detect the end of superclass initialization code. The last visitor is an annotation visitor, and it is used to parse DiSL snippet annotations.
The tree Api is used for everything else. DiSL implements a series of *Transformers* using the tree API, these transformers accept the list of instructions of a method plus some extra parameters, then the code is transformed and returned as an instruction list. Using this implementation, it is possible to chain multiple independent transformations one after the other.

The interface *CodeTransformer* shown in Figure 17 declares the methods *transform* and two versions for the method *apply*. The first method is going to be implemented by classes with specialized functionality, like in the example in Figure 18, the functions *apply* are used to chain multiple operations in order.

```
public interface CodeTransformer {

    abstract void transform (InsnList insns);

    static void apply(final InsnList insns,
        final CodeTransformer ... transformers) {
        Objects.requireNonNull(insns);
        Arrays.stream(transformers)
            .forEachOrdered(t -> t.transform (insns));
    }

    static void apply (final InsnList insns,
        final List <CodeTransformer> transformers) {
        Objects.requireNonNull (insns);
        Objects.requireNonNull (transformers).stream ().forEachOrdered (
            t -> t.transform (insns)
        );
    }
}
```

Figure 17. DiSL interface for Transformers, using ASM

The following example of a Transformer in Figure 18 takes a method and shifts all the local variable slots by an amount passed as a parameter in the constructor. The function *transform* edits every instruction that accesses the local variables.

```
final class ShiftLocalVarSlotCodeTransformer implements CodeTransformer {
    private final int __offset;

    public ShiftLocalVarSlotCodeTransformer (final int offset) {
        __offset = offset;
    }

    @Override
    public void transform (final InsnList insns) {
        Insns.asList (insns).stream ().forEach (insn -> {
            if (insn instanceof VarInsnNode) {
                ((VarInsnNode) insn).var += __offset;
            } else if (insn instanceof IincInsnNode) {
                ((IincInsnNode) insn).var += __offset;
            }
        });
    }
}
```

Figure 18. Transformer for shifting variable slots, using ASM

Applying a series of transformers is done in the following example in Figure 19.

```
final List<CodeTransformer> transformers = new ArrayList<>();
transformers.add(
    new ShiftLocalVarSlotCodeTransformer(-code.getParameterSlotCount())
);
if (options.contains(CodeOption.DYNAMIC_BYPASS) &&
    __snippetDynamicBypass) {
    transformers.add(new InsertDynamicBypassControlCodeTransformer());
}
if (options.contains(CodeOption.CATCH_EXCEPTIONS)) {
    transformers.add(
        new InsertExceptionHandlerCodeTransformer(
            __template.location(), code.getTryCatchBlocks()
        )
    );
}
final InsnList insns = code.getInstructions();
CodeTransformer.apply(insns, transformers);
```

Figure 19. Application of various transformers implemented in ASM

4.2.8 Class-File Traversal and Transformers

The thread-local variable visitor was replaced by using a *ClassBuilder* 3.4.2. When parsing the class java.lang.Thread every constructor method is discarded and replaced by a constructor with the same parameters. The code is edited to add the initialization code for each class variable. The function in Figure 20 takes three parameters: the list of local variables, the *ClassBuilder* that is currently being used for parsing the Thread class, and the original method, which is a constructor. Inside the function, the classBuilder is being used to invoke the method *withMethod*; this function adds a method to the class. The method generated is defined by the lambda passed as the fourth parameter. This lambda is a consumer that accepts a *MethodBuilder* 3.4.2. We can then iterate on the elements of the original method; elements that do not represent code are added through the methodBuilder unchanged. When we find the element that represents the method code, we invoke *withCode*, which also accepts a consumer function with type *CodeBuilder* 3.4.2. From this codeBuilder, we can finally add all the new initialization code before adding back all the original code.

The other two visitors were also changed; both the marker and the annotation parser were not refactored using Class-File transformers or builders.
In both cases, a simple parser iterating through the components was sufficient to replicate the effect of the visitors.

```
public static void insertThreadLocalVariables(
    final Set<ThreadLocalVar> threadLocals,
    ClassBuilder classBuilder,
    final MethodModel methodInitializer) {

    classBuilder.withMethod(methodInitializer.methodName(),
        methodInitializer.methodType(),
        methodInitializer.flags().flagsMask(),
        methodBuilder -> {
        for (MethodElement methodElement: methodInitializer) {
            if (methodElement instanceof CodeModel codeModel) {
                methodBuilder.withCode(codeBuilder -> {
                    for (final ThreadLocalVar tlv: threadLocals) {
                        codeBuilder.aload(0);  // this
                        loadInitialValue(codeBuilder, tlv);
                        putThreadField(codeBuilder, tlv); // put field
                    }
                    for (CodeElement codeElement: codeModel) {
                        codeBuilder.with(codeElement); // add original code
                    }
                });
            } else {
                methodBuilder.with(methodElement);
            }
        }
    });
}
```

Figure 20. Function to insert local variables using the Class-File API

The original DiSL transformers were changed to use a Class-File *MethodTransform* 3.4.2, this kind of element is a function that accept a *MethodBuilder* 3.4.2 and a *MethodElement* 3.4.2. The following code in Figure 21 is the same transformer presented in Figure 18, but using the Class-File.

```
public static MethodTransform shiftLocalVarSlotCodeTransformer(
    final int offset) {
    return (methodBuilder, methodElement) -> {
        if (methodElement instanceof CodeModel codeModel) {
            methodBuilder.transformCode(codeModel,
                (codeBuilder, codeElement) -> {
                switch (codeElement) {
                    case LoadInstruction loadInstruction -> {
                        final int newSlot =
                            Math.max(0, loadInstruction.slot() + offset);
                        LoadInstruction withOffset =
                            LoadInstruction.of(
                                loadInstruction.typeKind(), newSlot);
                        codeBuilder.with(withOffset);
                    }
                    case StoreInstruction storeInstruction -> {
                        int newSlot = Math.max(
                            0,storeInstruction.slot() + offset);
                        StoreInstruction withOffset =
                            StoreInstruction.of(
                                storeInstruction.typeKind(), newSlot);
                        codeBuilder.with(withOffset);
                    }
                    case DiscontinuedInstruction.RetInstruction
                        retInstruction -> {
                            DiscontinuedInstruction.RetInstruction withOffset =
                            DiscontinuedInstruction.RetInstruction.of(
                                retInstruction.opcode(),
                                retInstruction.slot() + offset);
                            codeBuilder.with(withOffset);
                    }
                    case IncrementInstruction incrementInstruction -> {
                        IncrementInstruction withOffset =
                            IncrementInstruction.of(
                                incrementInstruction.slot() + offset,
                                incrementInstruction.constant());
                        codeBuilder.with(withOffset);
                    }
                    default -> codeBuilder.with(codeElement);
                }
            });
        } else {
            methodBuilder.with(methodElement);
        }
    };
}
```

Figure 21. MethodTransform for shifting local variable slots with the Class-File API

This transformer is more verbose than the ASM counterpart; this is due to the subdivision of the Class-File as shown in Figure 16. ASM only has two nodes that interact with local slots, while in the Class-File, the instructions are divided into four different elements.
The other difference is that all elements passed as *CodeElement* inside the *methodBuilder.transformCode* are immutable, so the only way to change them is to create a new element with the slot value altered by the parameter *offset* of the function. The last difference is stylistic; the code is very indented and can become hard to read. This can often happen when using multiple transformers inside one another, like in this function, where a *CodeTransform* is declared inside the *MehodTransform*. The solution is to separate the transformers into different functions as well as to separate the code inside them into other static functions. An example of dividing a transformer was shown in Figure 11.

The following Figure 22 is an example of various transformations being applied; it is the Class-File equivalent of Figure 19.

```
final List<MethodTransform> transformers = new ArrayList<>();
transformers.add(
    ClassFileCodeTransformer
    .shiftLocalVarSlotCodeTransformer(-code.getParameterSlotCount()));
if (options.contains(CodeOption.DYNAMIC_BYPASS) && __snippetDynamicBypass) {
    transformers.add(
        ClassFileCodeTransformer.insertDynamicBypassControlCodeTransformer());
}
if (options.contains(CodeOption.CATCH_EXCEPTIONS)) {
    transformers.add(
        ClassFileCodeTransformer
        .InsertExceptionHandlerCodeTransformer(__template.location()));
}
MethodModel transformedMethod = ClassFileCodeTransformer
    .applyTransformers(code.getMethod(), transformers);
```

Figure 22. Application of various Class-File transformers

Class-File API offers functions to chain multiple transformers together. The function *andThen* takes a transformer and chains it with the transformer that invoked that method. The following example in Figure 23 is a simple way to chain a list of transformers using streams. The identity transformer used in the function *reduce* is a transformer that simply passes all the elements unchanged.

```
public static MethodTransform identityTransform = ClassFileBuilder::with;

public static MethodTransform applyTransformers(
    MethodModelCopy oldMethod,
    List<MethodTransform> transforms) {
    return transforms.stream()
        .reduce(identityTransform, MethodTransform::andThen);
}
```

Figure 23. Chaining transformers in the Class-File

4.3 Custom DiSL Context

In the original version of DiSL, users could extend a DiSL context to provide additional functionality. In these user-defined contexts, it was possible to obtain the list of instructions of the original method before being instrumented. The list and the contained instructions are represented with ASM data structures. This feature is still available in the Class-File version of DiSL, but the list and instructions use Class-File elements instead.
As a consequence, profilers that made use of user-defined context that included ASM operations are incompatible with the new DiSl. It is still possible to refactor the old profilers by replacing ASM with the Class-File API; this process is quite straightforward, but requires the user to manually perform the changes.

4.4 Missing Class-File Feature

Currently, the Class-File API does not offer any equivalent to the ASM analysis package, which allows performing symbolic execution on method instructions. In DiSL, this package is essential for implementing dynamic context functionality, optimization in the weaver, and more.
We decided to take the original ASM package and refactor it to use the Class-File API internally. This process was straightforward; most of the changes were required in the classes responsible for instruction interpretation. Otherwise, most of the classes inside the analysis package are not dependent on ASM instructions. To verify that the porting was correct, we also added unit testing.

4.5 Duplicated Instructions

In both ASM and the Class-File API, it is possible to parse a class and obtain the list of instructions for each method. Excluding pseudo-instructions, each element in the list represents an equivalent bytecode instruction. In ASM, the objects are uniquely represented in memory, and the same instruction in two different positions in the list is represented as an individual object in the JVM heap. This behavior is not replicated in the Class-File, for example, if we parse the following piece of code shown in Figure 24 as a list of instructions and if all the pseudo-instructions are removed, we obtain a list of five elements.

- Load[OP=ALOAD_0, slot=0]
- Invoke[OP=INVOKEVIRTUAL, m=C.j()V]
- Load[OP=ALOAD_0, slot=0]
- Invoke[OP=INVOKEVIRTUAL, m=C.j()V]
- Return[OP=RETURN]

```
public void m() {
    this.j();
    this.j();
}
```

Figure 24. Code of the five instructions

The list is composed of two load instructions for the receiver object, *this*, two invocations for the *j* function and a return. If the list is converted to a simple Java *Set*, we obtain the following elements.

- Load[OP=ALOAD_0, slot=0]
- Invoke[OP=INVOKEVIRTUAL, m=C.j()V]
- Invoke[OP=INVOKEVIRTUAL, m=C.j()V]
- Return[OP=RETURN]

Both load instructions were represented by the same object in memory, so the set only includes it once, while the invocations are two distinct objects, even if they represent the same invocation.
We discovered this behavior during testing since it was causing DiSL to crash with certain inputs. The simplest solution to this problem was to duplicate every real instruction in a new list while conserving all the pseudo-instructions, which guarantees that all newly created objects will be unique. The new list is then stored in a custom class that serves as a wrapper for the *MethodModel* class, but with the altered instructions list. Other potential implementations will be discussed in section 6.2.

4.6 LabelTarget and Label

The Class-File uses three elements to represent branching instructions. A *Label* 3.4.2 is a marker for a position inside a method, a *LabelTarget* 3.4.2 is a pseudo-instruction that holds a Label, and it implements the interface CodeElement, so it is found during the traversal of a method. The last component is *BranchInstruction*, which is the instruction responsible for branching. It represents a conditional or unconditional jump; the instruction contains an Opcode and a Label to point to the location of the jump. *TableSwitchInstruction* and *LookupSwitchInstruction* are instructions for representing a Java *switch*; they can access a jump table that includes Label objects for each target.
The Figure 25 shows the hierarchy between the elements; look-up switches and table switches are not included in the figure, but have the same position as *BranchInstruction*, with the difference that they can point to multiple *Label* objects.
Figure 26 shows the hierarchy of ASM branching for comparison.
Label and LabelTarget cannot be instantiated as other Class-File elements can; Label can only be created by a *CodeBuilder* or obtained while reading existing methods. LabelTarget cannot be created by the user at all; the intended way to bind a Label to a method position by the user is to invoke the virtual function *labelBinding(Label label)* on a CodeBuilder object. ASM pursues a similar structure for branching; it includes a class *Label* 3.4.1 that is equivalent to the Class-File. The class *LabelNode* 3.4.1 is a pseudo-instruction that accepts a Label and is the equivalent of LabelTarget. Both of these ASM classes can be created by the user using their public constructor. The last difference between ASM and the Class-File is that ASM jump instructions accept LabelNode objects instead of a Label. The DiSL (ASM version) weaver makes use of LabelNode objects to distinguish the various weaving regions that will have a snippet inserted. In this situation, the limitations of the Class-File pose a problem, the solution we found was to create a custom class named *FutureLabelTarget* that extend the abstract class *CustomAttribute* 3.4.2 which implement *CodeElement*. This allows our custom class to be mixed in lists and passed to functions that accept instructions and pseudo-instructions. Our custom class possesses two constructors, one that requires no parameters, which is used in the weaver, where we do not yet have access to a CodeBuilder. The other constructor accepts

a Label; this is used when a CodeBuilder is available just before inserting the snippet in the method. The reason is to replace the Label, the next section 4.7 will explain the reason for this operation.

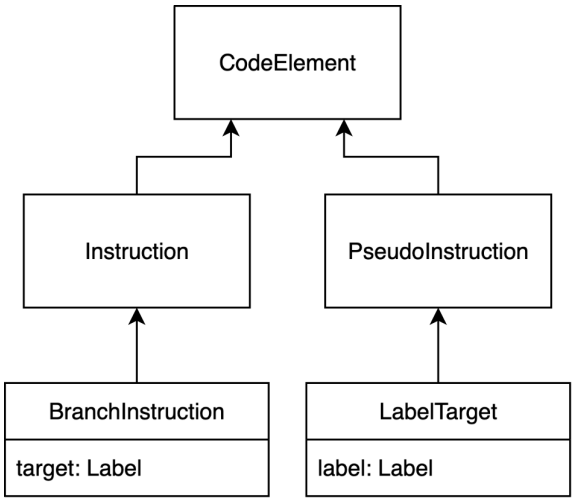


Figure 25. Hierarchy of the elements for Branching of the Class-File API

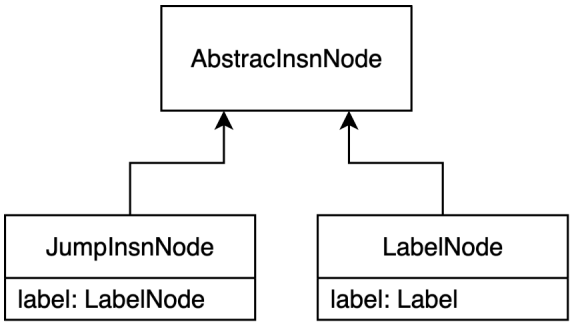
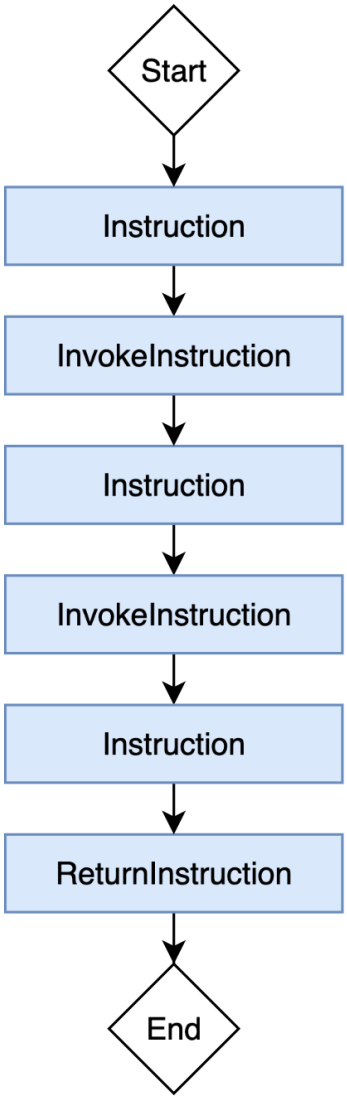


Figure 26. Hierarchy of the elements for Branching of ASM

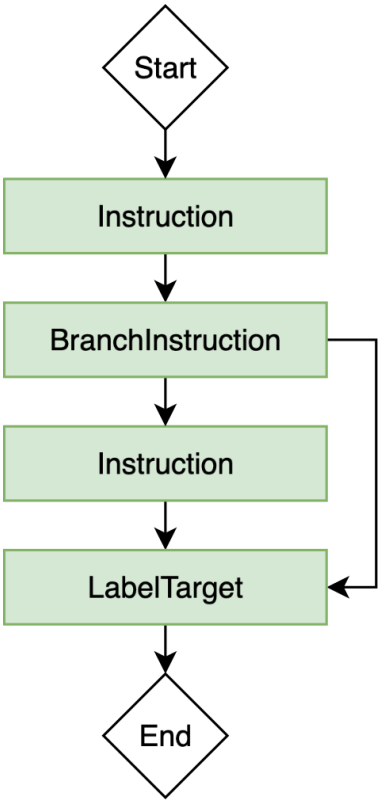
4.7 Multiple Snippets Insertion

When we recast the weaver, using the Class-File instead of ASM, we unintentionally introduced a problem that occurred only when inserting multiple snippets inside the same method. The bug occurred only if the snippet contained conditional or unconditional branching instructions, such as *GOTO*, *IFEQ*, or switches.

Figure 27a shows a control flow diagram for a simple method with two invocations. Figure 27b is the control flow diagram of a snippet with a branching instruction. Let's assume that we want to insert the snippet after every invocation of a method. The expected outcome is shown in Figure 28, the snippet is inserted twice, and every snippet is independent of the other.
Instead, the outcome of the instrumentation we obtained is displayed in Figure 29. In DiSL, when building an instrumented method, all the instructions are passed to a *CodeBuilder* sequentially; the order of each element passed determines the flow of the method. When passing a *LabelTarget*, the label is bound to the position, and all branch instructions that target the label will point to that position. If two identical labels are bound, the Class-File will only consider the last bind. This will cause the method to have an incorrect control-flow graph.
The solution to this problem is shown in Figure 30. We decided to replace all the branches and labels with new objects just before inserting the snippets. At this point of the instrumentation, we have access to the *CodeBuilder*, so it is possible to create new *Label* objects. We replaced all *LabelTarget* with our custom class *FutureLabelTarget* since *LabelTarget* cannot be instantiated by the user.



(a) Method instructions flow



(b) Snippet instructions flow

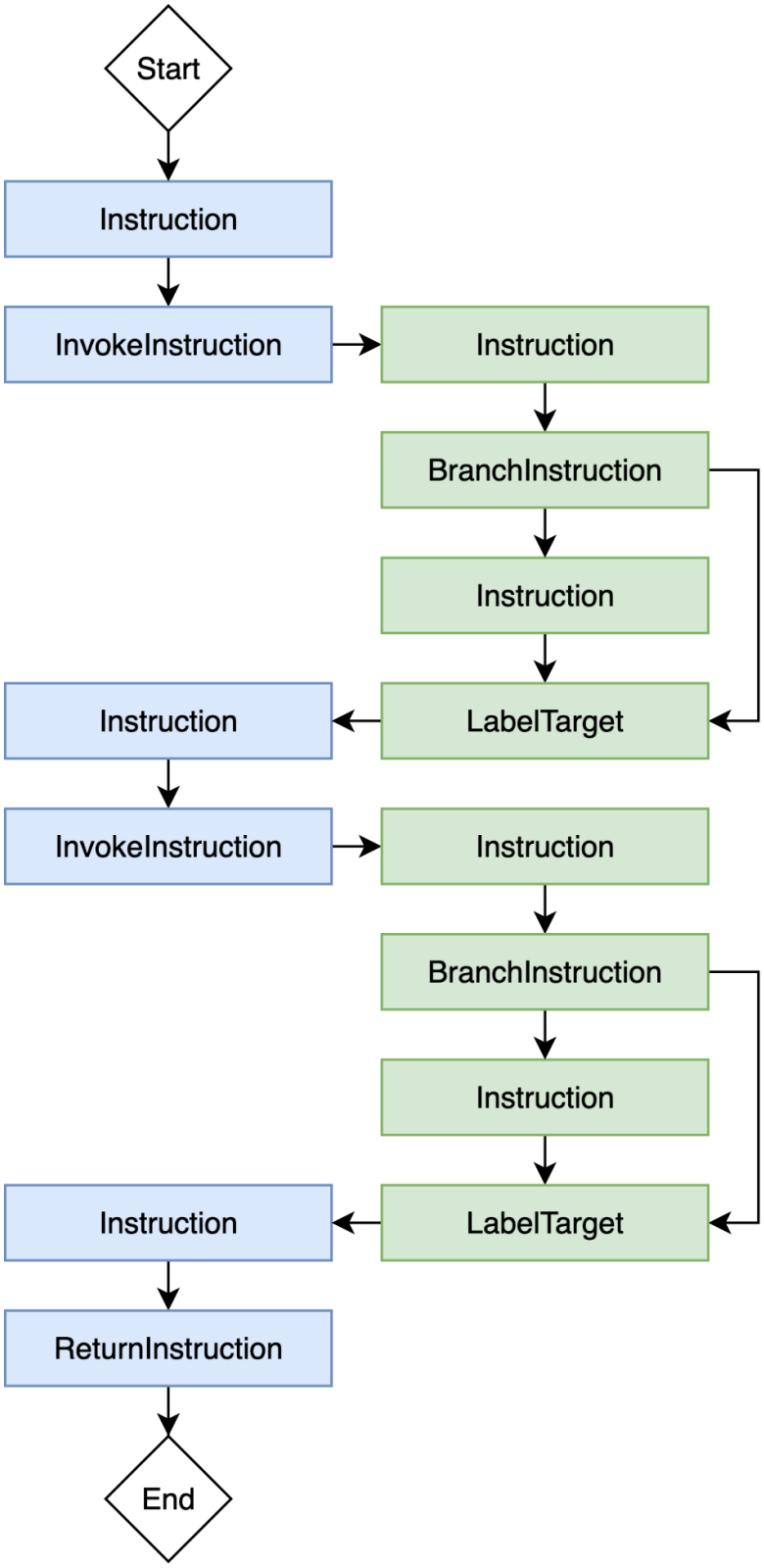


Figure 28. Expected instrumentation flow

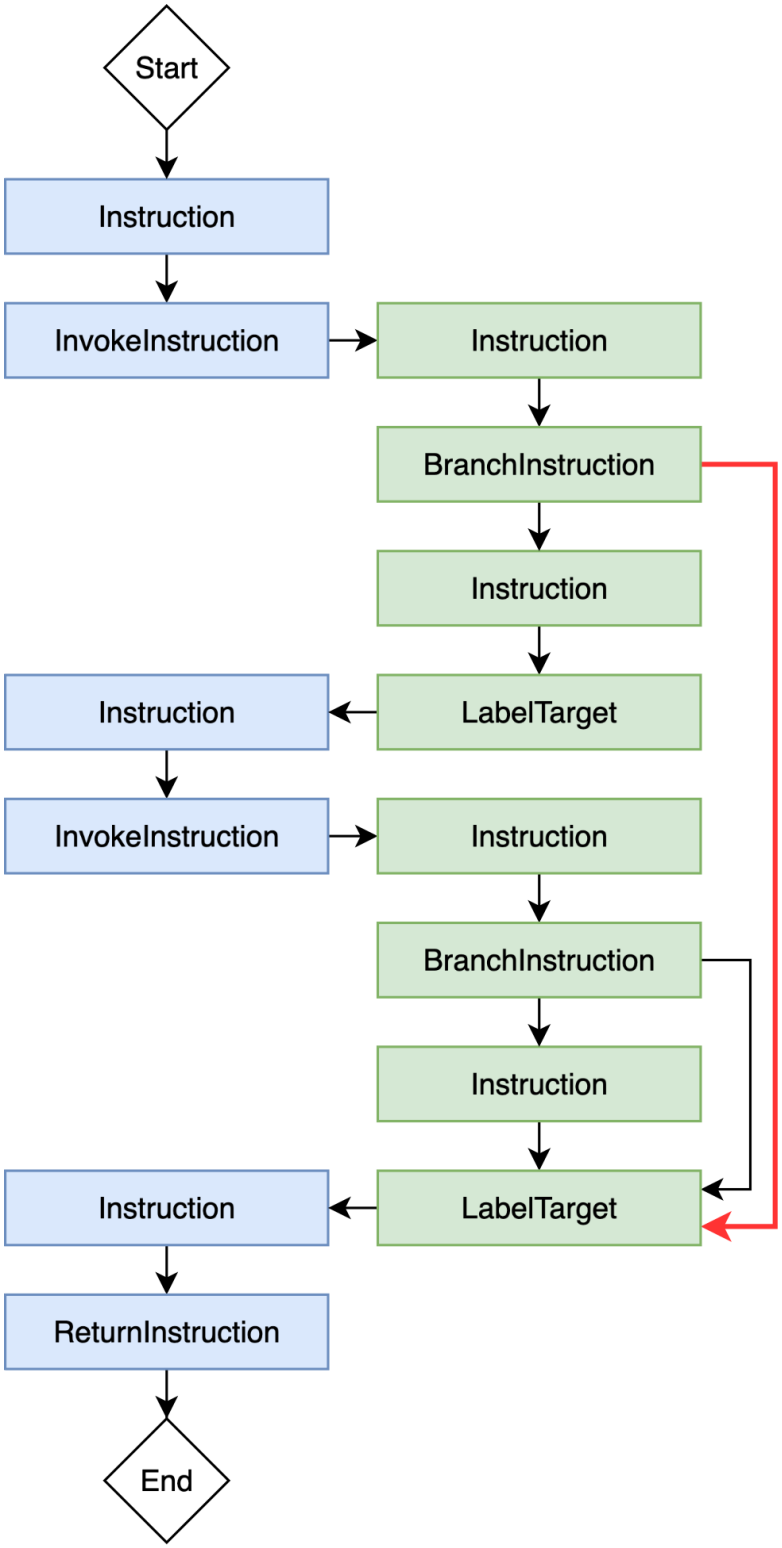


Figure 29. Incorrect instrumentation flow

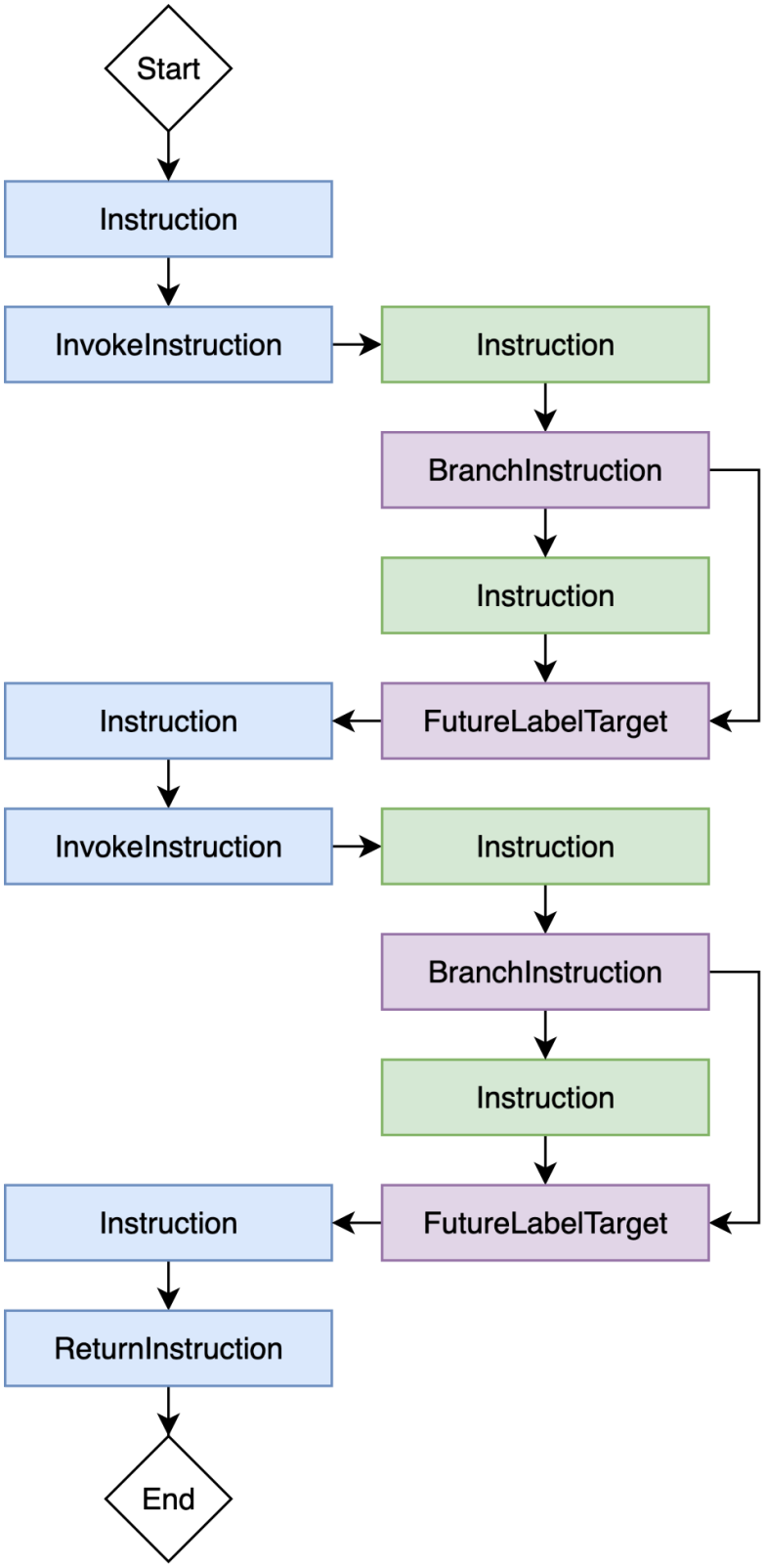


Figure 30. Branch and Labels of snippet replaced

4.8 Max Stack and Max Locals Calculation

We started recasting DiSL using JDK 23, and we switched to JDK 24 at release. In the preview version of the Class-File on JDK 23, every transformation automatically updated the max stack and max local values of the method for the next transformation. In JDK 24, with the full release of the API, this behavior was changed. The computation was deemed too expensive for each transformation, so only the first parsing of the class calculated the max stack and locals. Initially, we planned to use the Class-File API functionality to automatically calculate max stacks and locals, but we had to abandon the idea when upgrading the JDK. DiSL already includes functions to calculate the max stack and max locals using ASM, so we refactored them to use the Class-File instead.

4.9 Manual Class Loading

The Class-File API tries to internally resolve classes when transforming a class; this resolution is performed when calculating the stack map table. The Class-File will also verify the class when performing the transformation. DiSL does not have access to the class loader that the application being instrumented uses. This means instrumenting a class that uses a custom class loader might cause problems; in some cases, the Class-File will try to resolve a class that is not loaded into the DiSL process, since it is separated from the application, and it will throw an exception. This problem does not happen if the instrumented application was compiled with all the required classes.
We tried solving this problem by disabling the stack map generation, which fixes this problem, but another exception arises during the verification. We found another fix to the problem. If we use a *URLClassLoader* 3.4.3, which points to all the external classes, we can pass this loader to the Class-File, and when transforming the class, it will be used to resolve the classes. The Class-File will not crash since it can correctly resolve all classes. However, this is not a perfect solution as it requires manual pointing to the external classes. We believe that another, more practical solution could be implemented. The Class-File API specifies that it is possible to supply a stack map attribute when transforming a class. We could manually calculate a stack map and avoid the automatic resolution that the Class-File API performs. Due to time constraints, we decided not to pursue this solution further and to focus on the analysis.

4.10 ThreadLocal Variable Initialization

The original DiSL with ASM had some limitations when declaring thread-local variables. As Figure 31 shows, only primitives and constants like strings are allowed; objects and arrays must be initialized by declaring them with the value *null*.

```
@ThreadLocal
static int tlvInteger = 152; // valid, is a primitive

@ThreadLocal
static String tlvString = "My String"; // valid, is a constant

@ThreadLocal
static int[] tlvArray = new int[10]; // not valid

@ThreadLocal
static int[] tlvArray2 = null; // valid

@ThreadLocal
static List<String> tlvList = new ArrayList<>(); // not valid

@ThreadLocal
static List<String> tlvList2 = null; // valid
```

Figure 31. Thread local variables in ASM

This behavior is not caused by a limitation in ASM, but is due to the DiSL implementation of thread-local variables. The class *ThreadLocalVar* is used to represent a thread-local variable and has a single field to store the initial value. Additionally, the function responsible for parsing the snippet and setting the initial value for the variables only considers single instructions for initialization from the following list.

- ICONST
- LCONST
- FCONST
- DCONST
- BIPUSH
- SIPUSH
- LDC

Initialization for Objects and arrays requires more than a single instruction. When creating an array, it is also necessary to specify the size.
To expand this functionality while recasting DiSL, we changed the class *ThreadLocalVar* to store a list of instructions for initialization instead of a single value. The parsing function checks for additional instructions from the following list and saves all the necessary instructions that are required to initialize the variable. For example, the instruction *MULTIANEWARRAY* requires a set of integers for each of the dimensions.

- NEW
- NEWARRAY
- ANEWARRAY
- MULTIANEWARRAY

The list of instructions stored in *ThreadLocalVar* is later used when instrumenting the class *Thread*; all the instructions are inserted in the constructor, while the variable is added as a class variable.
The Figure 32 shows additional valid initialization in DiSL. Primitive arrays, object arrays, and objects created by a constructor are now valid. Initializations by calling static functions are not permitted.

```
@ThreadLocal
static int tlvInteger = 152; // valid, is a primitive

@ThreadLocal
static String tlvString = "My String"; // valid, is a constant

@ThreadLocal
static int[] tlvArray = new int[10]; // also valid

@ThreadLocal
static int[] tlvArray2 = null; // valid

@ThreadLocal
static List<String> tlvList = new ArrayList<>(); // also valid

@ThreadLocal
static List<String> tlvList2 = null; // valid

@ThreadLocal
static Object tlvFromFunction = staticFunction(10); // not valid
```

Figure 32. Thread local variables in the Class-File API

This addition can be extremely beneficial for performance in certain situations. The Figure 33 shows a snippet that requires the thread-local variable to be checked for *null* before being accessed. This check might not seem to have a strong impact, but it can add a large overhead if performed often. Figure 34 shows the same snippet, but with the thread local variable initialized once. The check for null can be removed, and no performance penalty is added.

```
@ThreadLocal
static List<String> tlvList = null;

@Before(marker=BytecodeMarker.class, args="invokevirtual", scope="*")
static void beforeEveryInvokeVirtual(CustomContext cc){
    if (tlvList == null) {
        tlvList = new ArrayList<>;
    }
    tlvList.add(cc.getInfo());
}
```

Figure 33. Snippet using the old initialization

```
@ThreadLocal
static List<String> tlvList = new ArrayList<>;

@Before(marker=BytecodeMarker.class, args="invokevirtual", scope="*")
static void beforeEveryInvokeVirtual(CustomContext cc){
    tlvList.add(cc.getInfo());
}
```

Figure 34. Snippet using the new initialization

This feature could also be added to the ASM version of DiSl by applying the same changes we introduced with the Class-File version.

4.11 Debugging and Testing

The only possible way to test DiSL during development was to perform unit testing on small helper static functions and independent modules, as opposed to integration testing. When the recast was completed, we were able to use the original DiSL tests.
Each test is composed of a snippet and a target class; they can be run individually to test each DiSL functionality in isolation. The test simulated a normal execution of DiSL using the agent and server. Initially, many of the tests crashed when used on the Class-File version of DiSL. To better debug them, we created a mock class that performed the instrumentation on a single class without using the server, which allowed us to attach a debugger to the process and follow the complete execution. The debugger saved us a lot of time. Once we managed to fix all the crashes, we compared the instrumented bytecode of the two versions of DiSL. We considered that the tests were successful when the instrumented bytecode was semantically equivalent, but not necessarily identical.
To further test DiSL, we used different simple profilers that were already used on the original DiSL. The difference between these profilers and the DiSL tests is that the profilers instrumented many classes at once, including classes in *Java.lang*. This time, we confronted the metric generated by the profilers instead of the bytecode. We asserted that the metrics were identical.
We decided to perform additional tests on a more complex profiler. The profiler in question instruments every polymorphic virtual method call to extract the information regarding the call-site and receiver type of the invocation. The objective is to find a mismatch between the profile *Hot Spot JVM* 3.4 uses for *C2* 3.4 compilation and the concrete behavior observed after the compilation.
Some modifications were required since the profiler uses ASM elements in a custom DiSL context, and it also needs to run on a modified version of JDK 21. We adapted the profiler to use the Class-File API in the custom context and removed some functions that required modification of the JDK since we needed to use JDK 24. The profiler was used with a simple test program. We confirmed that the metrics produced by this profiler with the Class-File version of DiSL were valid.
The last step in testing was to use DiSL with some real profilers on real benchmarks that were already proven on the original version of DiSL. The tests were performed on P^3 and focused on the performance; we explain in more detail and provide statistics in Section 5.1.

4.12 Changes Statistics

According to GitHub statistics, around 17000 lines of code were added and around 10000 were removed across 175 files.

5 Performance Evaluation

In this section, we present the experiments we performed and provide a timing comparison between DiSL with ASM and DiSL with the Class-File API.

5.1 Time Comparison on P^3

5.1.1 P^3

P^3 [9] is a profiler suite for parallel applications running on the Java Virtual Machine. It can profile metrics related to parallelism, concurrency, and synchronization, and it offers 15 different modules.

For this experiment, we used *ilock*, a module for profiling the use of implicit locks, namely synchronized methods and blocks. When using a module on selected benchmarks, P^3 performs multiple iterations. The user can specify this number, or if none is provided, the iteration will be stopped after a default number specific to each benchmark.

5.1.2 Renaissance Suite

Renaissance [10] is a benchmarking suite containing various popular systems, frameworks, and applications made for the Java Virtual Machine. The benchmarks offer a range of programming paradigms, such as concurrent, parallel, functional, and object-oriented programming.

5.1.3 Results

The experiment was performed using the following steps: each benchmark listed in the table 1 was executed ten times using the original DiSL with ASM and ten times with the Class-File API version of DiSL. All executions were performed in the same environment using the same settings with the module *ilock* from P^3.
The module was modified to avoid instrumenting the class *Thread*; the change was used on all executions with both versions of DiSL. The benchmarks are from *Renaissance* gpl version 0.16, and no modification was performed to them. We discarded some benchmarks that weren't working properly with the Class-File version of DiSL. Due to time constraints, we weren't able to investigate and fix the problems, so we chose to use only benchmarks that were working correctly.

When switching from a DiSL version to the other, we only replaced the *jar* libraries of DiSL. The Table 1 shows all the results from the experiment; the first column reports the name of the *Renaissance* benchmark. The second column indicates which iteration generated the results in the row; we chose to only consider the first iteration, represented by a zero, and the last iteration, represented by the non-zero value. This was done because the first iteration is also the only iteration that performs the instrumentation and is used as an approximation of the instrumentation time. While the last iteration is important, because the JVM will have optimized the most used methods. The third column reports the average time of the ten runs, which used DiSL with ASM, while the fourth column reports the same metric, but of runs that used the Class-File version of DiSL. The fifth column reports the ratio between the two previous columns. When the metric is greater than one, it means that the Class-File version performed better. The last column is the *p-value* between the two versions of DiSL, which was calculated

using the dependent t-test for paired samples, using a confidence level of 95%.

In most cases, the difference between the two versions of DiSL is not statistically significant; there are, however, some exceptions. The benchmarks *scala-doku* and *scala-kmeans* show that the ASM version performed better with a statistical significance only in the first iteration when instrumentation happens.

The other notable exception is in the benchmark *rx-scrabble*, where the Class-File version performed significantly better, especially in the last iteration. This difference could be explained by a JVM optimization, since the bytecode produced by both versions is semantically equivalent, but not identical; the Class-File version of DiSL might have instrumented the benchmark with a different branching flow, which the JVM was able to optimize. While in the ASM version, the JVM was not able to perform the same optimization.

We also computed the geometric mean of all the ratios of the first and last iteration, and we computed the median as well; the data are reported in Table 2.
The results are very close to one, and considering that most of the differences are not statistically significant, we believe that the ASM version and the Class-File version of DiSL perform similarly, and that the Class-File API is a valid alternative to ASM.

Benchmark	Iteration	Avg ASM (sec)	Avg Class-File (sec)	Ratio	p-val
akka-uct	0	7.234	7.014	1.031	0.086
akka-uct	23	5.778	5.674	1.018	0.551
fj-kmeans	0	2.453	2.492	0.984	0.07
fj-kmeans	29	2.295	2.325	0.987	0.219
future-genetic	0	1.196	1.196	1.0	0.97
future-genetic	49	0.97	0.977	0.993	0.386
mnemonics	0	1.676	1.682	0.997	0.687
mnemonics	15	1.548	1.565	0.989	0.28
par-mnemonics	0	1.442	1.469	0.981	0.238
par-mnemonics	15	1.322	1.325	0.998	0.929
philosophers	0	0.897	0.995	0.902	0.163
philosophers	29	0.494	0.49	1.009	0.789
rx-scrabble	0	0.487	0.304	1.602	1.373e-08
rx-scrabble	79	0.249	0.049	5.091	1.381e-13
scala-doku	0	1.788	1.82	0.983	0.019
scala-doku	19	1.272	1.287	0.989	0.378
scala-kmeans	0	0.193	0.206	0.937	0.038
scala-kmeans	49	0.119	0.125	0.951	0.27
scala-stm-bench7	0	1.045	1.096	0.954	0.083
scala-stm-bench7	59	0.505	0.499	1.011	0.606
scrabble	0	0.444	0.443	1.003	0.932
scrabble	49	0.204	0.208	0.98	0.517

Table 1. Results of Renaissance Benchmarks with P^3

	Geometric mean	Median
Ratio First Iteration	1.022	0.984
Ratio Last Iteration	1.151	0.993

Table 2. Geometric Mean and Median of the Ratio column

6 Discussion

In this section, we will list some of the limitations, as well will discuss possible future works that, due to time constrain, we weren't able to implement.

6.1 Limitation

6.1.1 Testing the Instrumentation Time

In our analysis, we used the first iteration to approximate the instrumentation time. The reason is that the very first iteration is the only one to perform instrumentation, while all subsequent iterations already have all the classes loaded in the JVM. The ideal measurement would have been to directly measure instrumentation time in DiSL. We avoided doing this because it would have been very challenging to correctly calculate the timing since the server accepts instrumentation requests while the application is running. We decided that approximating the result was acceptable to compare the two DiSL versions.

6.1.2 Compatibility with Older Profilers

To implement more complex functionality, many profilers make use of a custom context to access ASM instructions. This exact feature is not present in the recast version of DiSL by design; instead, the user must implement the custom context using Class-File API instructions. This means that older profilers that make use of a custom context need to be recast before being used in the Class-File version of DiSL.

6.1.3 Compatibility with Older JVM

The Class-File API was officially introduced in JDK 24; however, the API was already present since JDK 22, and could be used by enabling the preview features. From preview to full release, some of the API components were changed. We decided that we would support only the official release, meaning that the recast DiSL will only work with JDK 24 or newer.

6.1.4 Manual Class Loading

In specific cases, when the application we wish to instrument makes use of classes that are not loaded by the JVM default class loader, our Class-File version of DiSL requires the user to manually point the *jar* containing the external classes with a *URLClassLoader*. This problem does not present itself if the application is compiled with all the needed classes.

6.2 Alternative Approaches

An alternative we discussed was to use an internal DiSL representation for the bytecode instructions. The figure 35 shows how the process would look. First, the Java bytecode of the class to be instrumented is parsed using the Class-File API or ASM. Then it is parsed again to the internal DiSL representation. In this state, the methods are instrumented, and the snippet would also be first parsed in the same way and then inserted. Finally, the internal representation is converted back to either Class-File or ASM, and is then parsed back into instrumented bytecode for the Java Virtual Machine.
This approach has the benefit that the transformations and instrumentation would be completely independent of the library used to parse the bytecode. The downside is that we would need to develop the representation from scratch, as well as every other DiSL function would also need to be implemented again with all the helper functions. We decided to discard this

approach because it would have required an even larger refactor. Also, it would have likely taken longer to implement and test.

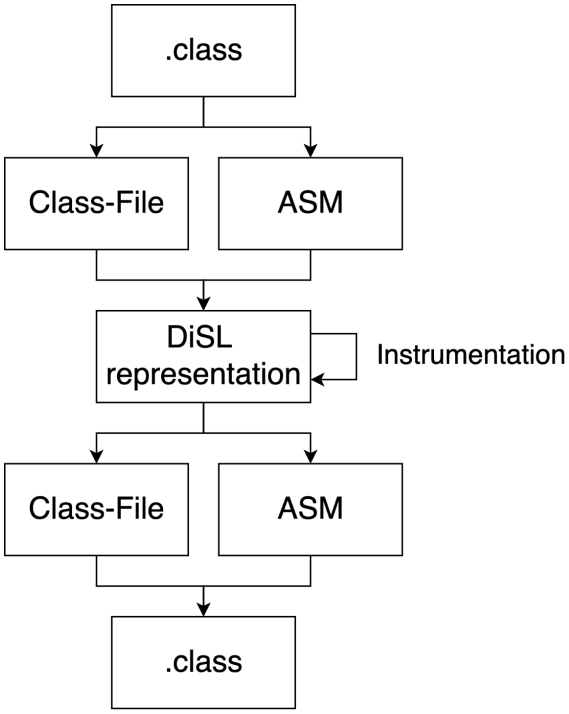


Figure 35. DiSL with an internal representation

Another potential alternative we discussed was the idea of creating more custom elements that could be inserted between the instructions to hold more information. This would have had the benefit that we could avoid re-creating all the instructions when first parsing a method. This approach was discarded since we already recast most of DiSL, and the implementations would have required a significant number of changes.

6.3 Future Work

Additional tests and analysis on the Class-File version of DiSL could be performed on different profilers and benchmarks to confirm our results.

The problem described in Section 4.9, where an applications that make use of custom class loaders could cause the Class-File to throw an exception, should be further investigated to implement a more convenient solution that does not require the user to perform additional work.

7 Conclusions

With our work, we refactored DiSL to use the Class-File API, which was recently released with JDK 24. We encountered many challenging problems during the recast, and we provided solutions as well as informative observations that might be useful when working with the Class-File API in other contexts than DiSL.
Finally, we performed an analysis comparing our version with the original DiSL using established profilers and benchmarks. From the result, we believe that the two versions of DiSL perform similarly, and that the Class-File API is a powerful and valid alternative to the ASM library.

References

[1] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: a domain-specific language for bytecode instrumentation. In *AOSD '12: Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 239–250, March 2012.

[2] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with aspectj. In *Software: Practice and Experience*, page 747–777, June 2007.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327 – 353, June 2001.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with aspectj. In *ACM*, page 59–65, 2001.

[5] Asm official website: https://asm.ow2.io/.

[6] Oracle official documentation on the jar command: https://docs.oracle.com/en/java/javase/24/docs/specs/man/jar.html.

[7] Oracle official documentation on the jlink command: https://docs.oracle.com/en/java/javase/24/docs/specs/man/jlink.html.

[8] Oracle. Jep 484: Class-file api. https://openjdk.org/jeps/484.

[9] Rosà Andrea and Binder Walter. P3: A profiler suite for parallel applications on the java virtual machine. In Bruno C. d. S. Oliveira, editor, *Programming Languages and Systems*, pages 364–372, Cham, 2020. Springer International Publishing.

[10] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the jvm. In *PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31 – 47, June 2019.

[11] Eugene Kuleshov. Using the asm framework to implement common java bytecode transformation patterns. In *AOSD.07*, March 2007.

[12] Romain Lenglet, Eric Bruneton, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, November 2002.

[13] Oracle. Openjdk official website: https://openjdk.org/.

[14] Oracle. Project nashorn official website: https://openjdk.org/projects/nashorn/.

[15] Apache software foundation. Apache groovy official website: http://www.groovy-lang.org/.

[16] JetBrains. Kotlin official website: https://kotlinlang.org/.

[17] Apache Common. Bcel official website: https://commons.apache.org/proper/commons-bcel/.

[18] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP 2000 - Object-Oriented Programming*, pages 364–376, 2000.

[19] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering*, pages 364–376, 2003.

[20] Rafael Winterhalter. Byte buddy official website: https://bytebuddy.net//.

[21] Cglib official repository: https://github.com/cglib/cglib.

[22] Spring official website: https://spring.io/.

[23] Hibernate official website: https://hibernate.org/.

[24] A. Abram White. Serp official website: https://serp.sourceforge.net/.

[25] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Mehmet Aksit and Satoshi Matsuoka, editors, ECOOP'97, volume 1241 of LNCS*, page 220–242, 1997.

[26] Walter Binder, Danilo Ansaloni, Alex Villazón, and Philipp Moret. Flexible and efficient profiling with aspect-oriented programming. In *Concurrency and Computation: Practice Experience, Volume 23, Issue 15*, pages 1749 – 1773, October 2011.

[27] Chukri Soueidi, Marius Monnier, Ali Kassem, and Yliès Falcone. Efficient and expressive bytecode-level instrumentation for java programs. In *International Journal on Software Tools for Technology Transfer*, page 453–479, June 2021.

[28] Chukri Soueidi, Ali Kassem, and Yliès Falcone. Bism: Bytecode-level instrumentation for software monitoring. In *Runtime Verification*, page 323–335, 2020.

[29] Luis Mastrangelo and Matthias Hauswirth. Jnif: Java native instrumentation framework. In *PPPJ '14: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 194 – 199, September 2014.

[30] David Georg Reichelt, Lubomír Bulej, Reiner Jung, and André van Hoorn. Overhead comparison of instrumentation frameworks. In *ICPE '24 Companion: Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, pages 249 – 256, May 2024.

[31] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. In *Journal of Software: Evolution and Process Volume 26, Issue 11: Mining Source Code Artifacts for Reverse Engineering - a special issue based on WCRE 2012*, pages 1030 – 1052, November 2014.

[32] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. A large-scale empirical study on java library migrations: Prevalence, trends, and rationales. In *ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 478 – 490, August 2021.