Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Design and Implementation of an Efficient Dynamic Slicer for Java

submitted by

Clemens Hammacher

on November 11, 2008

Supervisor

Andreas Zeller

Advisors

Valentin Dallmeier
Martin Burger

Reviewers

Andreas Zeller
Sebastian Hack

Affirmation of Congruence

Hereby I affirm that the content of the electronic version attached to this thesis conforms to this printed version.

Saarbrücken, November 11, 2008

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, November 11, 2008

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, November 11, 2008

Abstract

Dynamic slicing is a well known technique in automated debugging. For a concrete program run and the slicing criterion consisting of a program location and a set of variables, it computes the set of instructions that affected the values of the variables at this location.

Although dynamic slicing has been studied extensively, there is only one implementation of a dynamic slicer available for Java, called JSlice. Since this implementation does only work on few programs and is difficult to set up and to use, we are designing a new dynamic slicer (JAVASLICER) from scratch. To make our implementation future-proof and independent from the VM implementation, we use a Java Agent to instrument the bytecode of the program under suspect.

In this thesis, we describe the design decisions that have been taken as well as some aspects of the technical implementation. We also mention some improvements regarding the performance as well as the representation and compression of the execution trace, that builds the basis for the slice computation. In the evaluation section, we show that JAVASLICER is fully functional and exceeds the existing solution in several aspects.

Contents

List of Figures

List of Tables

Listings

1 Introduction

It is well known that debugging and maintenance take a great part in the development and lifecycle of a program. A current survey of 139 mid to big-sized North American companies conducted by IDC [3] during the 2nd quarter 2008 shows that each company spends $5.2 million to $22 million per year for debugging.

The question is what makes debugging that difficult, and hence that expensive. The problem is that in big companies, lots of developers work together on one software project. Each of them only works on a small part of the project, and has only a broad understanding of the internals of the other parts. So interacting with classes that another developer was responsible for may lead to the arise of errors in areas where one didn't expect them, because one didn't change anything there. The error might emerge as a wrong result in some computation, wrong behavior of the software or even a crash of the whole program. The challenge is now to find the source of the error, that is the component, and more precisely the instruction, that has to be changed in order to fix the bug. This task is the most time consuming one in debugging, after an erroneous behavior has been detected.

Beginners typically start by inserting special logging statements in areas where they expect the defect to be located. Their goal is to catch the control flow of the program and values of variables that they expect to have an influence on the bug. More advanced developers use debugging tools to place breakpoints into the same positions, which is already much better, but still has a big drawback: The developer has to understand the program in such detail, that she can tell what parts of the program could impact the observed bug, and has to *manually* set breakpoints, check the variables' values and analyze the control flow to finally find the cause of the error.

Even though this procedure might be feasible for small projects with only very few developers, it is very exhausting, because you rarely survey at first go the method where the defect is actually located. And so there is plenty of time spent on exploring dependencies in the source code to get an idea about all parts that may affect the bug.

In bigger projects these methods cannot be applied at all, because usually a single developer only has a deep understanding of a very limited part of the software, so it is impractical to manually analyze the dependencies and possible control flows of the whole program.

In the last 10 to 15 years, a lot of work has been done to automatize this first step of fixing a bug. So called *filtering techniques* [5] can be used to filter out code that cannot have influenced the becoming of the error. In this way, the amount of code that the developer has to explore to find the cause of the bug can be narrowed down.

The most important filtering technique is *slicing*: For any given program location and for any variable, it tells you which components affected the value of the variable in this program location. For Java programs, this *slice* - the result of the slicing algorithm - contains single bytecode instructions, so that you know very

precisely which parts of the source code you have to examine.

The slicing can be performed *statically* on the source code of the program, but because of special opportunities of object-oriented languages like dynamic binding, inheritance or polymorphism, the dependencies have to be resolved very conservatively in order to fetch all possibilities. Because of this constraints, the static slice is usually relatively big, and so barely helpful.

Much more precise is the *dynamic slice*, which is a subset of the static slice and is in most cases dramatically smaller, so it excludes much more code that cannot contain the origin of the error.

Slicing has been studied extensively for imperative programming languages like C, but there is no substantial progress for slicing object-oriented programming languages like Java. There exists only one working dynamic slicer for Java [6], but it is very outdated and difficult to use. So there obviously is the need for a modern and flexible dynamic slicer for Java programs, to contribute to more sophisticated automated debugging tools.

2 Related Work

There has been a lot of theoretical work regarding slicing in general, dynamic slicing of object-oriented programs specifically, and ways to store the execution trace in an efficient way.

We took many ideas from the work by Wang and Roychoudhury [11, 12], who found a great way of storing the execution trace, which is based on Java bytecode. They use lots of output streams that are separately compressed using an optimized version of the *SEQUITUR* algorithm developed by Nevill-Manning and Witten [9]. In this way they achieve a much better compression than if they would just write out the sequence of bytecode instructions as they are executed, and compress this single sequence. Their focus was mainly on an efficient representation of the execution trace, rather than an efficient slicing algorithm based on the recorded trace.

Other people regarded especially dynamic slicing on object-oriented programs: Zhao [14] as well as Xu and Chen [13] addressed the problems that arise from advanced features like inheritance, polymorphism or dynamic binding. They extended the concepts of the *dynamic dependence graph* and *control dependencies* to attain a precise dynamic slice.

But for our purposes, this is not necessary: By extending the definition of a *variable*, we can effortlessly use conventional slicing algorithms. Variables in object-oriented programs are all cells that carry independent information, like fields of an object, or each single entry of an array.

Concerning practical implementations of all this theoretical work, it looks very sparsely. To our knowledge, there exists only one dynamic slicer for Java, called JSlice [6]. It has been developed by Wang and Roychoudhury, and its first version is rather old. Since the mostly used Java VM by Sun was not published under an open-source license at that time, they decided to modify the Kaffe VM [7], a free clean-room implementation of the Java Virtual Machine as defined by Sun Microsystems. Kaffe is written completely in C, so they patched the source code to write out all information needed to reconstruct the trace of the program run. The version they used for their modifications is 1.0b3, which is mostly compatible with Sun-JDK 1.4. This old Java version is only rarely used today, but since the changes are very deep inside the system, it is hard to carry it over to a newer Kaffe version or even to another VM implementation.

The main disadvantage for users is that it is difficult to set up the Kaffe VM and JSlice for most systems. One has to compile it for every platform, and install it in parallel to the existing Java VM. The sourcecode they provide does only compile with the antiquated GCC version 2.95, and the binaries only run on some specific Linux distributions.

Its incompatibility with current Java versions makes it unusable for debugging newly developed software. For this reason, we chose the *DaCapo* benchmark suite [4] for evaluating our newly developed tool against JSlice. The DaCapo benchmarks consist of a collection of real life applications that have been selected according to several new metrics. These metrics provide for a diversity with respect

to code complexity, memory usage, object behavior and multithreading. Besides, this suite is compatible to JRE 1.4, which makes it usable with the Kaffe VM.

3 Background and Basics

3.1 Slicing

Program Slicing is a technique that is often used in automated debugging. As *filtering technique* [5], it can assist in narrowing down the source code statements that may have produced an error. It uses a well defined algorithm to find all statements that may have an influence on a user-defined variable, which is typically a variable that has been observed to hold an unexpected value. This form of slicing is called *backward slicing*. As this name implies, there is also a counterpart called *forward slicing*, which identifies all statements that are influenced in the future by a defined variable. Since forward slicing is rarely used, we only focus on backward slicing in this thesis.

Just like many other debugging techniques, slicing can either be performed statically (by analyzing the source code of the program), or dynamically (on an execution of the program with a specific input). For object-oriented programs, static slicing is quite complicated, since you have to deal with extended concepts like inheritance, polymorphism and dynamic binding. To capture all dependencies in the program, you have to be very conservative when analyzing method calls for instance. This fact often leads to very large static slices, that are not very useful in practice and can hardly be used in automated debugging to isolate possible error sources.

Dynamic slicing can compensate for most of these handicaps, since it actually knows the type of each object and which method is called by a statement. To have access to this information, it uses a *trace* of the program, that is recorded during its runtime. The computation of this trace is the most critical part in dynamic slicing, and takes a substantial part of this thesis. The resulting *dynamic slice* is usually dramatically smaller than the static slice, and is always a subset of it. This is obvious since the static slice contains all statements that *may* have influenced the suspected variables, whereas the dynamic slice only contains those statements that actually *did* influence them in this specific program run.

The *slicing criterion*, that is the only input of the slicing algorithm besides the program itself, consists of a location in the source code, as well as a set of variables referenced at this source code location. For dynamic slicing, the program location is given more precisely by stating a concrete execution of this location, usually given as the n'th or the n'th last execution of a certain statement.

Given the *dynamic slicing criterion*, the *execution trace* and the program itself, the dynamic slicing algorithm iterates backward through the execution trace to reconstruct the dynamic dependencies and in this way compute the dynamic slice. The static dependencies are defined as follows:

1. A *data dependency* from statement s to statement t by variable v exists, if and only if v is defined (written) in s and referred to (read) in t, and there is at least one execution path between s and t without a redefinition of v.

```
1  int gcd(int a, int b) {
2      while (a != b) {
3          if (a < b)
4              b -= a;
5          else
6              a -= b;
7      }
8      return a;
9  }
```

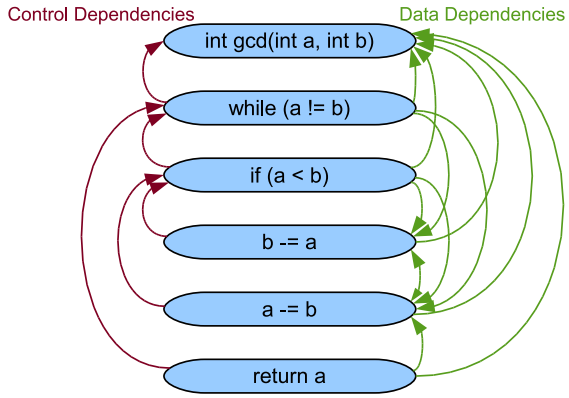

Figure 1: Illustration of data and control dependencies for a simple method

2. A *control dependency* from statement s to statement t exists if and only if s is a conditional predicate, for example the condition of an if- or while-loop, and the execution of t is controlled by s.

Figure 1 shows the static dependencies for a sample method.

The definition of *dynamic data dependencies* is straight forward since we know the exact path that has been taken by the concrete execution of the program. The *dynamic control dependencies* match their static counterparts, mapped to the concrete instances of the statements as they are executed.

The algorithm used to reconstruct these dependencies and to compute the slice out of them is described in section 5.5. In short, the backward slice just consists of the transitive closure of the dependencies starting at the variables and the location defined by the slicing criterion.

3.2 Java Bytecode

One of the main advantages of Java is its platform independence. It is achieved by not compiling the source code into machine code, that would only be executable on specific systems. Instead, Java uses an intermediate form called *bytecode*. On the target platform, this bytecode is loaded by the Java VM, and eventually translated into machine code that can directly be executed on the machine.

This characteristic of Java is also often seen as its major disadvantage, since it requires a *Java runtime environment* to be installed on the system that the program should be run on. But it also opens great opportunities for debugging, since the bytecode contains much more structural information than machine code. Besides, you can analyze and modify the bytecode, before it is loaded into the VM (see section 3.4).

The bytecode consists of more than one hundred different *bytecode instructions*, which look like some kind of assembler. Actually, bytecode instructions are a level higher than assembler, since almost all of them are translated into several

machine code instructions. The Java VM is based on a stack machine. It contains an internal operand stack, which the bytecode instructions use to load their operands from and to store results. It is also used to pass parameters to called functions: The parameters are just pushed onto the stack, then the method invocation instruction is executed.

In the following example you can see how the operand stack is used to compute an arithmetic expression. The method

```
1   private int divUp(final int a, final int b) {
2       return (a+b-1)/b;
3   }
```

is compiled to this bytecode:

```
1   ILOAD 1    // pushes the value of parameter 1 onto the stack
2   ILOAD 2    // pushes the value of parameter 2 onto the stack
3   IADD       // pops the top two elements off the stack
4              // and pushes the sum of them back on the stack
5   ICONST_1   // pushes the constant 1 onto the stack
6   ISUB       // pops the top two elements off the stack,
7              // subtracts the second-top from the top element and
8              // pushes the result back on the stack
9   ILOAD 2    // pushes the value of parameter 2 onto the stack
10  IDIV       // pops the top two elements off the stack,
11             // divides the second-top by the top element and
12             // pushes the result back on the stack
13  IRETURN    // pops the top element off the stack and returns it
```

In this short example you can see that the method parameters a and b are referred to as the local variables 1 and 2, and the prepended I indicates that integer operations are to be performed.

During the *translation* (also called *compilation*) of the Java source code into bytecode, a *.class*-file is generated for each single Java class, including nested and anonymous classes. These class files are often packed into a *.jar*-archive, which can easier be delivered to the end-user. This archive is just a zip file containing all class files plus some special meta information (like the *main class*, whose main method should be called if the jar file is *executed*).

3.3 ASM

ASM [2] is a fast and powerful bytecode analysis and manipulation framework. It provides a more usable representation of the bytecode than just the binary code that is contained in the class file. Each class can be parsed into an object-oriented representation, which can easily be modified using classes included in ASM, and written out again as binary data to a class file. The format of the class file was defined by Sun Microsystems, and is described very precisely by Lindholm and Yellin [8].

Besides the raw conversion and presentation of bytecode instructions, ASM inserts a *label* at each bytecode offset that is the target of a jump instruction, an

Listing 1: Simple method with try-catch block

```
1  private Object arrayGet(final Object[] a, final int i) {
2      try {
3          return a[i];
4      } catch (final ArrayIndexOutOfBoundsException e) {
5          return null;
6      }
7  }
```

Listing 2: Bytecode corresponding to the method in Listing 1

```
1  // access flags 2
2  private arrayGet([Ljava/lang/Object;I)Ljava/lang/Object;
3    TRYCATCHBLOCK L0 L1 L2 java/lang/ArrayIndexOutOfBoundsException
4     L0
5      ALOAD 1
6      ILOAD 2
7      AALOAD
8     L1
9      ARETURN
10    L2
11     ASTORE 3
12     ACONST_NULL
13     ARETURN
14   MAXSTACK = 2
15   MAXLOCALS = 4
```

exception handler or the beginning of a new line number (if the compiler was told to include debugging information in the class file).

These label objects are used in the corresponding jump instruction or try-catch-block definition to refer to that jump target. An example of how this looks like in ASM's bytecode presentation is given in Listing 2.

ASM provides two different interfaces for analyzing and manipulating Java bytecode: Either using the *visitor pattern* to present the elements to the user while reading them from the class file, which is the fastest and most efficient way. The other possibility is to let ASM read in the whole class and examine or manipulate it in a tree structure. This second way is slower since the whole class first has to be read in and held in memory, then the user has the possibility to work on it as he wants, and possibly afterwards the whole class has to be written out again. This way should only be chosen if the manipulations are too complex to be performed by a visitor, or if you need information about elements that appear later in the class file. Obviously, in this second presentation, the user has much more freedom to inspect the bytecode by iterating through the methods, the instructions of the methods, or other information like try-catch-blocks or annotations in any direction and insert, remove or change elements.

In Listing 3, we show how the bytecode of Listing 1 can be produced using

Listing 3: ASM code producing the bytecode of Listing 2

```
1  ClassWriter cw = new ClassWriter(0);
2
3  cw.visit(V1_6, ACC_PUBLIC + ACC_SUPER, "de/unisb/cs/st/A", null,
4      "java/lang/Object", null);
5
6  MethodVisitor mv = cw.visitMethod(ACC_PRIVATE, "arrayGet",
7      "([Ljava/lang/Object;I)Ljava/lang/Object;", null, null);
8  mv.visitCode();
9  Label l0 = new Label();
10 Label l1 = new Label();
11 Label l2 = new Label();
12 mv.visitTryCatchBlock(l0, l1, l2,
13     "java/lang/ArrayIndexOutOfBoundsException");
14 mv.visitLabel(l0);
15 mv.visitVarInsn(ALOAD, 1);
16 mv.visitVarInsn(ILOAD, 2);
17 mv.visitInsn(AALOAD);
18 mv.visitLabel(l1);
19 mv.visitInsn(ARETURN);
20 mv.visitLabel(l2);
21 mv.visitVarInsn(ASTORE, 3);
22 mv.visitInsn(ACONST_NULL);
23 mv.visitInsn(ARETURN);
24 mv.visitMaxs(2, 4);
25 mv.visitEnd();
26
27 cw.visitEnd();
28
29 byte[] classBytecode = cw.toByteArray();
```

ASM and the visitor pattern.

3.4 Java Agents

A Java Agent is a bundle of Java classes that is able to manipulate classes as they are loaded into the Java VM - and since Java 1.5 even afterwards.

To understand where in the VM the Java Agent attaches, we have to take a look at how classes are loaded into the VM. Each class is loaded by so called class loaders on the first time that it is needed (*on demand*). The queried class loader is responsible for finding the class's bytecode and passing it to the VM.

After start-up of the VM, there is only one class loader: the *system class loader*. It uses the classpath that has been set for the Java process to locate class files in the local file system or inside jar archives.

Once the core system of Java has started, arbitrarily complex class loaders can be defined, which load the class definitions over a network for example, or generate the classes' bytecode on the fly.

Class loaders can even be nested in an arbitrary depth, and can cooperate to

load a class.

When the bytecode is available - regardless of which class loader found or generated it - one classloader calls the method `defineClass` to give the new class definition to the VM.

Now this is the time where Java Agents come into play: Before the VM interprets or compiles the bytecode to machine code, it passes it through the *instrumentation* framework. Java Agents, that are declared by command line arguments of the Java VM, have the possibility to register *class file transformers* with the instrumentation framework. Each of these transformers will get the chance to manipulate the bytecode of the class being loaded. There are no restrictions how the transformers can modify the class: They can add, change, or remove methods, fields, implemented interfaces, annotations, and everything else that defines a Java class.

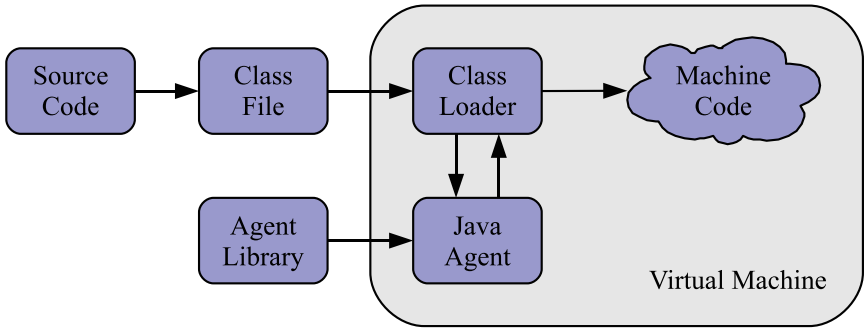

Figure 2: Sketch of the Java toolchain

Starting from Java 1.5, it is even possible to *redefine* (or *retransform*) classes that are already loaded into the VM. There is only a small difference between a *redefinition* and a *retransformation* of a class: A redefinition completely replaces a previous definition of the class, while a retransformation just modifies some pieces of the class. Nevertheless, (at least) up to Java 1.7, there are some restrictions when redefining or retransforming a class: These procedures may change method bodies, the constant pool or attributes, but it must not add, change or remove fields or methods or change inheritance of the class.

3.5 Sequitur

Sequitur is a compression algorithm worked out by Nevill-Manning and Witten [9]. The basic idea is to build a *context-free grammar*, that accepts exactly the sequence that has been compressed. If the sequence contains many repetitions, this would result in a very good compression rate.

The algorithm starts with a grammar containing exactly one rule that assigns the empty sequence to the start symbol. Whenever a new symbol is appended to the sequence, it is just appended to the rule defining the start symbol. Then the

grammar is adjusted to conform to the following invariants:

1. No pair of adjacent symbols appears more than once in the grammar
2. Every rule (except for the one defining the start symbol) is used more than once

Using the term *digram*, which is defined as two adjacent symbols occurring at least once in the sequence, we can restate the first invariant as 'every digram in the grammar is unique'.

The algorithm is designed to exactly ensure these invariants. It achieves this by maintaining a set of already seen digrams, and for every new digram that occurs (either by appending a symbol to the start symbol's rule, or by restructuring rules), it checks whether it is already contained in the set of known digrams. If this is the case, it replaces both occurrences (the one stored in the set of known digrams and the new one) by a new non-terminal symbol, that refers to a newly built rule mapped to the digram that has been replaced. If the duplicate digram matches exactly the body of a rule in the grammar, then of course no new rule has to be generated, but the non-terminal can just refer to the rule that maps to that digram. In this way, a rule can be referenced more than twice.

This procedure is recursive, since after restructuring, there are up to four new digrams that have to be checked: For each of the locations where the new non-terminal was inserted, there could be one digram including the previous symbol and the new one, and one including the new one and its successor. One starting at the symbol preceding the first inserted non-terminal, one starting at that non-terminal, and the same for the second non-terminal.

Whenever a new rule is built, it may happen that another rule is exclusively used in the body of the new rule. In this case, the non-terminal referring to this uniquely used rule is replaced by the body of that rule and the rule is removed from the grammar.

Table 1 shows an example of how the algorithm works on a small sample string.

3.5.1 Runlength Encoding

If the grammar contains many long repetitions of the same terminal or non-terminal symbol, the original Sequitur algorithm cannot compress them in an efficient way.

Consider the string containing n times the constant c. The grammar that is built by the algorithm to match this string will consist of $log(n)$ rules. Using a small enhancement, this drawback can be removed: If every terminal and non-terminal symbol has an additional *count* parameter, the string can be represented by just one symbol.

This enhancement was described by Wang and Roychoudhury [11], and we took it over to our Sequitur implementation.

New Symbol	Grammar
	$S \rightarrow \epsilon$
1	$S \rightarrow 1$
2	$S \rightarrow 1\,2$
3	$S \rightarrow 1\,2\,3$
4	$S \rightarrow 1\,2\,3\,4$
2	$S \rightarrow 1\,2\,3\,4\,2$
3	$S \rightarrow 1\,2\,3\,4\,2\,3$
	$S \rightarrow 1\,A\,4\,A$ $A \rightarrow 2\,3$
4	$S \rightarrow 1\,A\,4\,A\,4$ $A \rightarrow 2\,3$
	$S \rightarrow 1\,B\,B$ $A \rightarrow 2\,3$ $B \rightarrow A\,4$
	$S \rightarrow 1\,B\,B$ $B \rightarrow 2\,3\,4$

Table 1: Example of the Sequitur algorithm compressing a sample sequence of integers

3.5.2 Shared Grammar

Normally, Sequitur is used to compress one sequence of symbols (the *input string*), and the grammar constructed by the algorithm is used as a compressed representation of the input string.

As described in section 4.1.3, the tracer generates a lot of sequences, under which many are very similar. When compressing them, their grammar will also be similar. To avoid writing this redundant information into the trace file, we modify Sequitur in such a way, that all sequences share the same grammar (which now contains several start symbols). This global grammar is usually dramatically smaller than the sum of all separate grammars. In particular, if there are several identical strings, they together do not need more space than just one of them, since they refer to the same start symbol. The situation of identical or at least very similar sequences occurs very often in tracing, for example if writing to several object fields in a method. Consider for example the following method:

```
1 private void set(int val) {
2   this.a = val;
3   this.b = 2*this.a;
4   if (this.a != this.b)
5     this.c = 0;
6 }
```

In this method, there are six sequences that store object identities: One in line 2, two in line 3 (one for the read and one for the write), two in line 4, and one in line

5. The first five sequences are always identical, since the reference to *this* cannot change. The sixth sequence will probably be very similar to the others, since the object identity is only omitted if the parameter *val* is zero. So these six sequences do not need much more space than just the first one alone.

This modification introduces only one slight drawback: The grammar now has to be stored separately from the compressed strings, so these are not independent any more. When reading in the compressed sequences, we first have to read in the grammar, and then we pass a reference to the already read grammar for reading in the sequences.

4 Design

4.1 Tracer

Tracing the execution of the program is the most complicated and most time consuming part of dynamic slicing. So for designing the tracer, a major challenge is efficiency in runtime, memory and disk space used to collect and store the trace.

4.1.1 Instrumentation

The concept of Java agents has already been introduced in Section 3.4.

We use a Java agent to instrument the bytecode of all classes that are currently loaded into the VM, and new classes as they are loaded. This includes classes contained in the JRE, since it is not sufficient to just instrument user classes. If a user program uses a `Vector` for example, we have to know which instruction wrote which single element in the vector in order to be able to compute a precise slice. The instrumentation of these *standard library classes* introduces some problems that are discussed in Section 5.2.

The instrumentation procedure inserts new bytecode instructions before or after certain instructions, so that we can write out a file (called *trace file*), which contains all information needed to compute the dynamic slices on this program run that the user may request.

The information that the slicer needs is the precise sequence of bytecode instructions that have been executed, as well as some dynamic information as discussed below. So we can split up tracing in two parts: tracing the control flow, and tracing dynamic data.

For tracing the control flow of the execution, we have to consider all possibilities for jumps in the control flow, where a *jump* denotes the contiguous execution of two instructions that are not direct neighbors in the bytecode.

There are several ways how jumps can occur: The most obvious way are unconditional jump instructions (goto, jsr) and conditional branches (ifnull, ifle, if_implt, ...). The next and more complicated instructions are method invocations. Java uses four different bytecode instructions for method invocation: Two of them make use of *dynamic binding* (invokeinterface, invokevirtual), the other two do always refer to the same method (invokestatic, invokespecial). Because of dynamic binding, it is very complicated to find out the instruction succeeding the method invocation. We would have to dynamically find out the type of the object on which the method is invoked (a reference to that object is located on the internal operand stack), and resolve the method call manually, which is very time consuming.

Another cause of jumps are exceptions. They can not only occur at athrow instructions, but also on any instruction that accesses an object through a reference that it reads from the operand stack, for example array or field accesses, or method invocations. These instructions would throw a `NullPointerException` if the reference is `null`. Other instructions throw `ClassCastExcep-`

tions, ArrayIndexOutOfBoundsExceptions or ArithmeticExceptions, so there are a lot of locations that may throw exceptions where is it not obvious.

So the naive approach of tracing whenever a jump takes place (or could take place) is very hard to implement, and very time consuming during the execution of the program.

Since we are only interested in the backward traversal of the execution trace, we take another approach. It is based on the premise that if the control flow changes, it always continues at a label. This is obvious for jumps and conditional branches, since their argument is a label. But also every catch block starts with a label, and since finally blocks are translated into catch blocks that catch every Throwable, they are covered as well. Nevertheless, we have to take care of method invocations. In order to trace them adequately, we insert additional labels at the beginning of each method body (to catch the jump into the method body), and after each method invocation instruction (to catch the return from a method). The special bytecode instructions jsr and ret, that call local subroutines inside the same method, and are used by the Java compiler to implement finally blocks, are just handled like ordinary method calls, so they introduce no additional difficulties here.

With these tricks, our premise is correct, and the tracing during the execution of the program in order to be able to reconstruct the control flow is very easy: Every time we cross a label, we store the *index* of the bytecode instruction that we come from (this is the bytecode instruction that was executed just before the label was crossed). The indexes are assigned statically to the instructions during the instrumentation. This is sufficient in order to enable us to reconstruct the sequence of executed bytecode instructions (the *control flow*) from the trace file.

But for dynamic slicing, we need some more information: Since for every variable in the program, the algorithm needs to know when it is read and when it is written, we have to trace the identity of the involved array during array-load or array-store instructions, and the identity of the object whenever one of its fields is read or written. The object reference that is used by these instructions is located on Java's operand stack. In front of each of these instructions, we add some bytecode instructions that get the object reference off the stack and trace its identity (as long value) to the trace file. See Section 4.1.2 for the details of how the long values are assigned to the objects.

This covers all accesses to shared state, but still there is a problem with local variables: They are just accessed by their index inside the method being executed, but in order to resolve the dependencies correctly, we have to know the stack frame in which the instructions are being executed. Under normal circumstances it is easy to simulate the stack frames: Whenever a method is entered, we push a "fresh" stack frame onto the stack, and when a method is left, we pop one off the stack. Once again, the concept of exceptions breaks this beautiful simplicity. When an exception is thrown, we have to trace through how many stack frames it falls until it is catched.

To do so, we put a try-catch block around each method body that catches any

exception, traces this event, and rethrows the exception. With this information we can reconstruct the method frame stack depth for any instruction of the execution trace.

4.1.2 Identifying Objects

The tracing algorithm often needs to write out the *identity* of an object. Java already provides something similar, called the *hashcode* of an object. The hashcode is an integer value that can be accessed through the `Object.hashCode()` and `System.identityHashCode()` methods. But for our purposes, it is not good enough: If two objects are semantically identical (if they are *equal*), the JRE defines that they should yield the same hashcode. But even in this case, we want to distinguish the two objects, since their fields may have been written by different instructions, and the objects may later change so that they are not equal any more. And even if two objects are not equal, they could have the same hashcode, either due to weaknesses in the hashcode computation, or more fundamentally because there are only 2^{32} different hashcodes.

So we define the requirements of object *identities* as follows:

1. The identity of an object does not change over time.
2. Two distinct objects do never have the same identity.

To achieve these requirements, we have to use long values (64-bit integers) as data type, because even in relatively short programs, there can be more than 2^{32} objects. This is boosted by Java's *autoboxing* feature, which makes it easy to convert between primitive types like `int` or `byte` and their corresponding wrapper classes `Integer` and `Byte`. But since programmers often do not recognize when this autoboxing is performed, it leads to the creation of many short-living objects, which nevertheless get their own identities assigned.

The assignment and access of identity values have to be very efficient, since it is done frequently during tracing. The most efficient way would be to add a new field to the `Object` class that holds the identity value of each object. This field would be initialized in the `Object` constructor (which is called by any constructor), and would not change afterwards. But due to the restrictions of retransforming already loaded classes (see Section 3.4), this is not possible, since Object is one of the first classes that is loaded by the VM.

So we decided to maintain a map, that assigns each object its corresponding identity value. Of course this map has to be implemented in such a way that objects that are not needed anymore are removed from this map. Fortunately, Java provides a great way of implementing such maps, that still allow the garbage collector to clear the objects if they are not referenced elsewhere. The way to do this are *weak references*, through which the object can be accessed as long as it exists in memory, and when the garbage collector recognizes that the object is only reachable through weak references, it reclaims it and notifies us about that.

The JRE contains an implementation of a hashmap that uses weak references to reference its keys, but on the basis of our described requirements, we decided to use a special combination of the three classes `WeakHashMap`, `IdentityHashMap` and `ConcurrentHashMap`, which all only cover one part of the requirements.

A performance related improvement, that evades this map in certain cases, is described in Section 5.4.

4.1.3 Trace File

All information fetched during instrumentation and during the run of the instrumented program is written into one file. This information contains a representation of all classes, including their bytecode instructions, as well as all traced information.

The traced information is split in many distinct *sequences*, since each instrumented location in the bytecode generates its own sequence during runtime. The sequences do either contain an integer value (for the backward instruction pointer at labels, and for array indexes) or a long value (to identify objects). In order to be able to read each of these sequences individually when reading in the traced information, each sequence is written to a separate *stream*.

Since it would be very inefficient and unhandy to write each stream to a separate file, all streams are written into just one file. Therefore, the trace file implements a specialized *virtual filesystem* (it is more a virtual streamsystem actually), where several streams can be read and written simultaneously. The file is split in blocks, each of which is assigned to at most one stream. Some details of the implementation and performance issues are discussed in Section 5.4.

4.1.4 Compression

Since the trace file grows rapidly during the runtime of the instrumented program, it makes sense to compress it in some way. We implemented several compression algorithms, that can be selected by the user. They all have one property in common: They do not compress the whole trace file, but only the streams contained in it. This is necessary because during slicing, we only need to traverse some of the sequences stored in the trace file, and it would be inefficient to first decompress the whole file.

Another requirement is backward-iterability: During slicing we iterate backward through the execution trace. But since during the execution of the program, the sequences are generated and written out in the order in which they occur, the compression algorithm used must either support backward-iterability natively, or we have to reverse and compress each sequence when it is finished, which is not very efficient.

A comparison of the different compression algorithms regarding their runtime and compression efficiency can be found in Section 6.3.1.

4.2 Slicer

For dynamic slicing we are basically using the algorithm presented by Wang and Roychoudhury [11, 12]. But since this algorithm is not designed for object-oriented programs, we have to extend the definition of a variable. In pure imperative programming languages there are two kinds of variables: *Global* variables can be accessed from within any method and by any thread (they are *shared* between all threads). In contrast, *local* variables can only be accessed from within the method they are declared in. Both memory locations can hold *array* types: data structures, in which several elements are grouped together.

In object-oriented languages like Java, there is one more location where elements can be stored: in the fields of an object, which is stored on the *heap*. Besides, global variables are encapsulated in classes, and are called *static* fields. So in object-oriented languages, a variable as for slicing is either a local variable, a static field, a field of an object, or an element of an array.

With this definition of a variable, Wang and Roychoudhury's algorithm can easily be adapted for our purposes. See Section 5.5 for a description of the precise implementation and Section 5.3 for the limitations of this approach.

Listing 4: Simple class implementing the lazy initializing pattern

```
1  public class InstrumentationExample {
2
3    public static class Helper { }
4
5    private Helper helper;
6
7    private Helper getHelper() {
8      if (this.helper == null)
9        this.helper = new Helper();
10     return this.helper;
11   }
12 }
```

5 Implementation

5.1 Instrumentation

In Section 4.1.1, we already described briefly where new instructions are added in the bytecode, and what information they record. In this section, we will show in detail how the instrumented bytecode looks like and how the generation of the trace file works.

We start with an example of a naively instrumented simple method without any effort to increase performance. Consider the small class shown in Listing 4, which consists of a lazy-initialized field and a getter-method.

In bytecode, this method already has 11 instructions. Since the instrumentation increases the method's body to 105 instructions, we only show a decompiled version of the instrumented code (see Listing 5).

The first thing to notice is that `Tracer` is a singleton class, whose instance can be obtained by the static method `Tracer.getInstance()`. It is also evident that each thread has an associated `ThreadTracer` object, that is obtained by the method `getThreadTracer()` on the `Tracer` object.

The instrumented method code starts in line 8 with writing the index of the last executed instruction to the stream 1000 (assuming that at the time that this method was instrumented, there were 1000 streams and 4000 instructions in previously instrumented classes). This index is stored inside the `ThreadTracer` object, and is changed to `i` by a call to `passInstruction(i)`.

The index of the last executed instruction is written out at the beginning of each method, and is needed when reconstructing the backward trace of the program: When the algorithm reaches the beginning of the method, the next integer in stream 1000 tells him where to continue (which instruction called this method).

In line 9, a try/catch block starts as described in Section 4.1.1. If any exception occurs, the catch block starting in line 24 first writes out the index of the instruction at which the exception was raised and then passes the special instruction 4003. During the reconstruction of the trace, this instruction tells the algorithm that the

Listing 5: Instrumented version of the class in Listing 4

```
1  public class InstrumentationExample {
2
3      public static class Helper { }
4
5      private Helper helper;
6
7      private Helper getHelper() {
8          /* abbreviated as getThreadTracer():
9             Tracer.getInstance().getThreadTracer(). */
10         getThreadTracer().traceLastInstructionIndex(1000);
11         try {
12             getThreadTracer().traceObject(this, 1001);
13             getThreadTracer().passInstruction(4000);
14             if (this.helper == null) {
15                 getThreadTracer().passInstruction(4001);
16                 Helper tmp = new Helper();
17                 getThreadTracer().traceLastInstructionIndex(1002);
18                 getThreadTracer().traceObject(this, 1003);
19                 getThreadTracer().passInstruction(4002);
20                 this.helper = tmp;
21             }
22             getThreadTracer().traceLastInstructionIndex(1004);
23             getThreadTracer().passInstruction(4002);
24             return this.helper;
25         } catch (final Throwable t) {
26             getThreadTracer().traceLastInstructionIndex(1005);
27             getThreadTracer().passInstruction(4003);
28             throw t;
29         }
30     }
31 }
```

method was not exited normally by a return statement, but that an exception terminated the execution of the method.

Inside the body of the try/catch block, the instrumented code of the original method reemerges. Line 10 and 11 correspond to the statement in line 12, which contains a getfield instruction. For this getfield instruction, the identity of the object on which it is performed is written out into sequence number 1001. After that, the `ThreadTracer` is notified that instruction number 4000 is about to be executed. This call to the method `passInstruction` is inserted before every single bytecode instruction, so in bytecode, there are much more than in this simplified example.

In line 14, a new `Helper` object is created. This contains a call to the default constructor of the `Helper` class. After this method call, again the last executed instruction (a return statement of the `Helper.<init>()` method) is written to the sequence 1002, just as after any other method call.

Afterwards, the identity of `this` is written to sequence 1003, because in line

18, a putfield operation is performed on that object.

The last interesting point is line 20. Because this is a jump target of statement 12, we have to trace the last executed instruction, to know whether we came from instruction 4002 or 4000.

5.2 Challenges

During implementation some major challenges appeared that took a lot of time to cope with.

The first one arises from the instrumentation of classes contained in the JRE. The necessity of instrumenting these classes was discussed in Section 4.1.1. Unfortunately, some of these classes are also used in our implementation of the tracer, so we have to ensure that we only trace code that has been invoked by the user program, and not by the tracer itself. Ignoring this would not only result in incorrect slices, but it would also result in stack overflows during runtime.

The solution to this is to pause tracing whenever entering a tracing method, and continuing when the method is left.

5.3 Limitations

Unfortunately, there are some limitations of our implementation of the tracer and slicer. Some of them are general for every tracer that is based on a Java agent, others are specific to our implementation.

5.3.1 Native code

In Java methods can be declared as *native*, to indicate that the method body is not implemented in Java. Instead, it is loaded from a dynamically linked library and called by the JNI (the *Java Native Interface*). These methods are implemented in C, C++, or in raw assembly to evade limitations of the Java language definition, to use a library written in C or C++, to speed up critical sections of a software product, or to call system-specific functions, that are not available through the Java API.

This native code cannot be accessed and modified by the class file transformers, so they cannot be instrumented and hence cannot be traced. If the native code changes any values, be it static fields, field values in objects, or array elements, we do not see this. So this leads to imprecise traces, resulting in imprecise slices.

If native code calls another method that is implemented in Java, this method call is recognized by the tracer, but it does not know where the call comes from. The tracer misses all stack frames that belong to native methods, so if we refer to the top two elements of the reconstructed frame stack as A and B, the slicer cannot tell whether A called B directly, or whether one or several native methods were involved between them.

In most Java projects, native code is rarely used, since it breaks the platform-independence of Java and is difficult to use and to debug. But there are many classes in the standard library of Java (the JRE), that use native code either to directly access resources managed by the operating system (such as I/O access or sound capabilities), to speed up execution (for instance in `System.arraycopy`), or to make direct use of processor instructions to implement methods like `AtomicInteger.compareAndSet`. Even though the first case is not that critical for slicing, the other two are, since we lose data dependencies here.

One possibility to limit the flaws introduced by native methods would be to identify the most important parts of the JRE that are implemented natively, and during slicing, whenever one of these methods is called, we update the dependencies manually. This includes analyzing the implementation of the native methods and hardcode the dependencies in the slicing algorithm. This has not been done for this thesis, so it stays future work.

5.3.2 Reflection

In Java, a program can examine its own structure using a technique called *reflection*. This provides for example the ability to get a list of all fields or methods of a class, to call methods, and to get or modify the values of object fields. Since the low-level parts of this framework are implemented using native code (see Section 5.3.1), these accesses cannot be traced. This constraint leads to imprecise slices, but reflection is only rarely used, especially in critical parts of the software.

5.3.3 Shutdown hooks

Shutdown hooks are threads that can be registered with the `Runtime` object by the Java program during its execution. They are started when all non-daemon threads have finished, thus when the Java VM otherwise would terminate. The termination is postponed until all shutdown hooks have finished execution.

Since we use a shutdown hook to stop tracing, finish and write out the trace file, it is possible to miss information about the execution of other shutdown hooks.

A possible workaround would be to wait for all other non-daemon shutdown hooks to finish before stopping tracing, but this is not that easy: We do not know, what other shutdown hooks exist, since they are started one after the other, and even if we would solve this, we could provoke deadlocks if other (user-)threads do the same.

So we decided to constrain tracing so that we do not guarantee whether shutdown hooks are traced.

5.4 Performance Tuning

Since our implementation of the tracer should be usable in practice, the main goal during design of the tracer was to make it easy to use by developers who are testing

their software product. In the implementation step, the main problem - beside ensuring correctness - is to make the tracer as efficient as possible. Unfortunately, the first version of the tracer had an overhead of more than 1000, so the execution time increased by a factor of at least 1000 when the tracing Java agent was added. It is quite evident that it is necessary to improve the efficiency by reducing the overhead introduced by the instructions that were added during instrumentation. On the next pages, we will show what we have done in order to achieve a feasible efficiency.

Since the runtime overhead of the tracing agent is dependent on the program that is executed and on the input, we only state the relative improvement of each optimization that is discussed below.

The first approach to reduce the runtime overhead is to reduce the amount of inserted bytecode instructions. Like described in Section 4.1.1, we insert instructions at each label that occurs in the bytecode, in order to trace the control flow. It turns out that ASM adds a lot of labels that are not used as jump targets or markers for starting catch blocks. They are used for example to associate source code line numbers to the bytecode instructions, or as start marker for a try block. So for these labels, we do not have to trace the index of the previously executed instruction, since this will always be the instruction right in front of it.

Beside this, the additional labels added at the beginning of each method and after each method invocation are sometimes not necessary, since there already is a label in exactly that position. The removal of all these labels resulted in a decrease of the tracing overhead to 90%.

Another improvement aiming at the avoidance of unnecessary method calls is very obvious when looking at the example in Section 5.1. The call to `Tracer.getInstance().getThreadTracer()` appears at every inserted instruction and is very expensive. It results in a map lookup that has to be synchronized with other threads, so we should try to avoid as many of these calls as possible. The method `getThreadTracer()` returns a `ThreadTracer` object, that is responsible for all tracing actions of the particular Thread executing the current method. So in one Thread, this call will always return the same object. Hence it is possible to get this object only once at the beginning of each method, and store it in a local variable. This reduces the overhead greatly to 4.5%.

To even avoid the call at the beginning of each method, we tried to pass the `ThreadTracer` object between methods. To do so, we duplicated each method, and added an additional parameter to the copied method, that gets the `ThreadTracer` object. Unfortunately, this is only possible for newly loaded classes, since in retransformation, it is not allowed to add methods. So if we modify the method invoking instruction by adding the additional parameter, we have to ensure that every class that this call could be bound to could be modified by adding new methods. Since we cannot statically assure this for invokeinterface and invokevirtual, we can only use this improvement for private, non-native methods. With a restriction to these method calls, an effect on efficiency could not be measured.

To further reduce the number of calls to the `ThreadTracer` object, we could

determine bytecode instructions whose successor during execution is always the successor instruction in bytecode. These are all instructions that do never jump to a label, and cannot throw any exception. For these instructions, it is not necessary to call the method `passInstruction()` which stores the index of the instruction into the `ThreadTracer` object, since in any case the next instruction is executed afterwards, which overwrites this information.

When implementing this, it effectively reduces the size of the classes that are produced, but it does not decrease the runtime at all. It turns out that the hotspot optimizer contained in Sun's Java VM already throws these instructions away, that do not have any effect on the later execution of the instrumented method.

The optimizations described so far do all aim at reducing the amount of instructions that are inserted during instrumentation. Now we want to discuss some approaches that are based on heuristic assumptions or other considerations, that are not straight intuitive.

As described in Section 5.2, all tracing methods (`passInstruction()`, `traceInt()`, ...) first check whether the `ThreadTracer` is currently paused. If this is the case, they have no effect. When analyzing when the paused flag is set and reset, it turns out that if it is set at the beginning of one method, it will be set during the whole method. So we could evaluate the paused flag once at the entry of each method, and leave the tracing method calls out, if the flag is set. To achieve this in the most efficient way, the original (uninstrumented) bytecode of each method is copied at its end, after a special label. At the beginning of each method, we immediately jump to this label, if the `ThreadTracer` is currently paused. In this way we save all the tracing method calls, which improves the performance by about 3%.

An even better way to save the overhead of the instrumented bytecode is to use special classes in all tracing-related methods, which are not instrumented. In general, all classes contained in packages of the tracer are not instrumented, because this would immediately lead to a loop, and hence a stack overflow. So by replacing some frequently used classes like `DataOutputStream` or `ArrayList` by own implementations, which are not instrumented, we could save another 66% of the introduced overhead.

The last low-level optimization concerns the writing of all traced information into one single file. For this job, we wrote a `MultiplexedFileWriter` class, which implements a stream based virtual file system, where each file (or *stream*) is accessed by an integer identifier. With this class, several thread can write simultaneously to different streams, which are all stored in one file. Like other file systems, the file is organized in blocks, and each stream consists of a collection of blocks in this file.

By caching each block and only writing it out when it is full, we could linearize the write operations to the file, so that there are no seeks necessary - new blocks are just appended to the file. This reduces the runtime by 30%, and by using memory-mapped files, it can be reduced by another 10 to 20%. With memory-mapped files, large pieces of the file are mapped directly into the (virtual) main memory

of the running Java process by the operating system, and when the process writes to this memory sections, the operating system is responsible for eventually writing back these changes to the file. This memory-mapping is only feasible on 64 bit platforms, since on 32 bit, the virtual address space of the size 2^{32} bytes (= 4 gigabytes) is too small to map large pieces of the file in there.

After all these optimization steps, the runtime of a program increases by the factor of 30 for tracing into an uncompressed file, and 40 if compressing using *gzip*.

5.5 Slicing Algorithm

Our algorithm used for computing the slices is based on the one presented by Wang and Roychoudhury [11, 12]. The main challenge that remained was the computation of control dependencies. Since the dynamic intra-procedural control dependencies do relate to the static control dependencies mapped to the dynamic instances of the instructions, it is useful to precompute the static intra-procedural control dependencies.

For this computation, we build the complete *control flow graph* for each method, and store a mapping that assigns each instruction the set of all instructions depending on this one. This computation takes some time, but since it is only done once per method, this has no negative consequences for the overall efficiency.

Listing 6: Demonstration of a simple slice

```
1  public class MethodSlicing {
2
3      public static void main(String[] args) {
4          int a = 1;
5          int b = 2;
6          int c = getFirst(a, b);
7          return;        slicing criterion: c
8      }
9
10     private static int getFirst(int first, int second) {
11         return first;
12     }
13 }
```

6 Evaluation

The evaluation of our tool, which we called JAVASLICER, is performed in several steps: To show the correctness of the algorithms used, we first compute traces and slices for small pieces of code (Sections 6.1.1 and 6.1.2). After that, we apply JAVASLICER to some of the programs contained in the *DaCapo* benchmark suite, which has been introduced in Section 2.

6.1 Case Studies

6.1.1 Assignments and Methods

Our first case study includes simple assignments and a method call (see Listing 6). We want to check whether this method call is traced correctly, and whether the slicer computes the correct data dependencies. To be able to observe this, we chose the local variable `c` at line 7 as the slicing criterion. This allows us to detect all statements, that led to the assignment of `c` in line 6. In Listing 6, the slicing criterion has been marked in light grey.

In line 6, there are data dependencies between the local variables `a` and `b` and the parameters of the method `getFirst`. Another data dependency exists between the return statement in line 11, and the assignment of variable `c` in line 6. To detect these dependencies, the slicer already has to examine the operand stack. The values of `a` and `b` are pushed onto the stack, then `getFirst` is called, and after the method call returns, its result is popped off the stack and stored in the local variable c.

Since the method `getFirst` does not use its second parameter, the value of c on line 6 should only depend on the assignment of variable a in line 4, the method call in line 6, and the body of the method in line 11.

6.1.2 Exception Handling and Advanced Data Dependencies

This case study includes throwing exceptions, controlled by variable values, and catch blocks (see Listing 7). The challenge for the tracer is to correctly trace the control flow, which is disarranged by the abnormal abortion of the method execution. We expect the slicer to compute the control dependencies in such precision that it finds out why the exception has been thrown, and where the exception message comes from. To complicate this task, we define the error message as a static field, which gets passed to the constructor of the `NullPointerException`. After several super-constructor calls, it is finally stored in the `detailMessage` field of the class `Throwable`. In line 16, the method `getMessage()` reads it from there, and assigns it to the local variable `error`, which makes up the slicing criterion.

As shown in Listing 7, the slicer determines that only variable `b` is responsible for the `NullPointerException`, and that the error message that is assigned at line 16 depends on the assignment in line 8.

6.2 Real Life Applications

After these first case studies which show that our implementation works in general, we apply it to some of the real life applications contained in the DaCapo suite.

In order to demonstrate the correctness of the computed slices, we intend to run both our implementation and JSlice on the same program using the same slicing criterion, and compare the results. The first remark is that JSlice does only output the name of the source code file and the line number for each instruction contained in the slice. JAVASLICER outputs each precise bytecode instruction. But since the bytecode instructions can easily be mapped to their line numbers, we can compare the results at line level.

Unfortunately, it turns out that JSlice does not work with any program contained in the DaCapo suite. There is always a `NullPointerException` thrown before the actual test program starts running. Hence, we have nothing to check our computed slices against. We checked a few of them manually, but this is only feasible for very small slices. There is no possibility to ensure that our slicer works for bigger slices, even if we have confidence in that.

6.3 Performance

Since a major issue concerning the usability of JAVASLICER is performance, we spent a lot of work in this (see Section 5.4). In order to proof that the tool is efficient enough to be used in daily debugging, we conducted performance tests regarding the runtime of the tracing and the slicing component, as well as the size of the trace file.

Listing 7: Slice computation including exceptions

```
1  public class ExceptionSlicing {
2
3      private static String ERROR_A;
4      private static String ERROR_B;
5
6      public static void main(String[] args) {
7          ERROR_A = "a is null";
8          ERROR_B = "b is null";
9          int[] a = new int[1];
10         int[] b = null;
11         String error = null;
12
13         try {
14             addLengths(a, b);
15         } catch (NullPointerException e) {
16             error = e.getMessage();
17         }
18
19         return;        slicing criterion: error
20     }
21
22     private static int addLengths(int[] a, int[] b) {
23         if (a == null)
24             throw new NullPointerException(ERROR_A);
25         if (b == null)
26             throw new NullPointerException(ERROR_B);
27         return a.length + b.length;
28     }
29 }
```

6.3.1 Compression

As described in Section 4.1.4, we provide different compression algorithms, that the user can select when creating the trace file:

- *no compression*: Each sequence is written to a separate stream in the trace file, using 4 byte per integer and 8 bytes per long value. In this way, the stream can easily be iterated backwards.
- *gzip*: Each sequence is compressed individually using the common gzip algorithm. Since gzip does *not* provide backward-iterability, we compress the reversed sequence. To save memory, we temporarily store the sequence in a stream of the trace file just as for *no compression*, but when the sequence is finished, we read it from behind, compress it, and write it into another stream. After that, the uncompressed stream is removed, freeing the blocks in the trace file that it consumed.

		Traced Execution (s)			Trace File Size (MB)		
Subject	Runtime (s)	Raw	GZip	Sequitur	Raw	GZip	Sequitur
antlr	0.8	23.2	35.6	36.1	429.2	11.3	2.8
bloat	1.8	36.9	57.7	103.8	614.5	19.5	8.5
fop	6.0	21.8	25.8	33.1	178.2	14.3	3.6
hsqldb	1.1	27.0	40.3	48.8	403.9	71.2	5.3
jython	6.1	98.6	237.9	233.3	2834.8	24.7	8.7
luindex	2.4	36.5	63.6	91.0	874.6	13.2	5.8
lusearch	1.5	87.4	162.2	175.8	2989.6	28.6	10.2
pmd	1.2	15.8	17.0	18.0	47.1	12.1	2.8
xalan	2.6	94.0	203.4	193.1	2522.5	106.7	29.6

Table 2: Comparison of different compression algorithms. Sequitur produces much smaller files than gzip in acceptable execution times

- *Sequitur*: The Sequitur algorithm was introduced in Section 3.5. The major drawback of this algorithm is that the whole compressed sequence has to be hold in memory, since whenever a digram occurs, which has already been used in an arbitrary position in the sequence before, it has to be replaced by a new rule. But as shown in Table 2, this investment of memory and runtime can pay off in most cases: Sequitur does always yield better compression results than gzip, sometimes the size of the resulting trace file is less than 10 % of the size produced by gzip.

Table 2 shows a comparison of the different compression algorithms, as well as the respective runtime overhead. All tests have been performed on a quad-core machine (4×2.4 GHz) with 8 GB of main memory.

It is obvious that using no compression yields the best execution times, but often produces huge trace files. Due to the organization of the trace file as described in Section 4.1.3, gzip and Sequitur both result in excellent compression rates. In most cases, Sequitur does only need slightly more time than gzip, but produces trace file of one third to one tenth of the size.

6.3.2 Slicing

Measuring the slicing performance is not that simple since the execution time strongly depends on the selection of the slicing criterion. To reduce this impact as far as possible, we perform several hundred slice computations on randomly chosen slicing criteria.

The results of this test series are visualized in Figure 3. It is clearly visible that there is a strong correlation between the resulting slice size and the runtime of the slicing algorithm. We had to exclude the two test programs *jython* and *xalan* from this graph, since their execution times were much higher and hence not representable in this diagram.

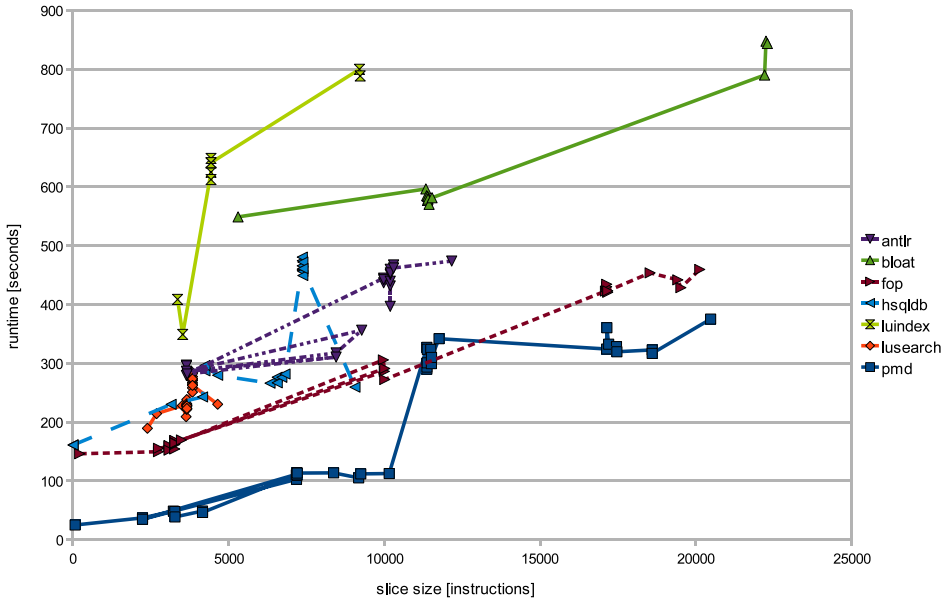

Figure 3: Comparison of the slicer runtime against the size of the resulting slice shows a linear correlation

Figure 4 shows a weak relation between the length of the execution trace and the average time consumed for a slice computation. *Jython* has by far the longest execution trace (about 1.7 billion instruction), but it fits in the row of the other programs, since it also has a much higher runtime of slicing on average.

The only program with a completely different behavior is the *xalan* program. Although it has a very short trace length of about 19 million instructions, the time needed to compute slices on this trace is very high. This discrepancy already showed up in Table 2, where *xalan* has by far the largest compressed trace file sizes. These curious observations can be partially explained when looking at the peculiarities of *xalan* [4]. Under all programs contained in the DaCapo suite, it uses the most heap space and allocates the most objects. Its complex memory usage patterns seem to increase the size of the trace file while likewise increasing the slicing runtime.

6.4 Comparison with JSlice

Comparing our tool to JSlice brings up some major differences:

- *Usability*:
  As objected before, JAVASLICER is much easier to deal with than JSlice. There is only one thing a developer has to do to write out the execution trace of a program: Specifying the Java Agent along with some parameters at the

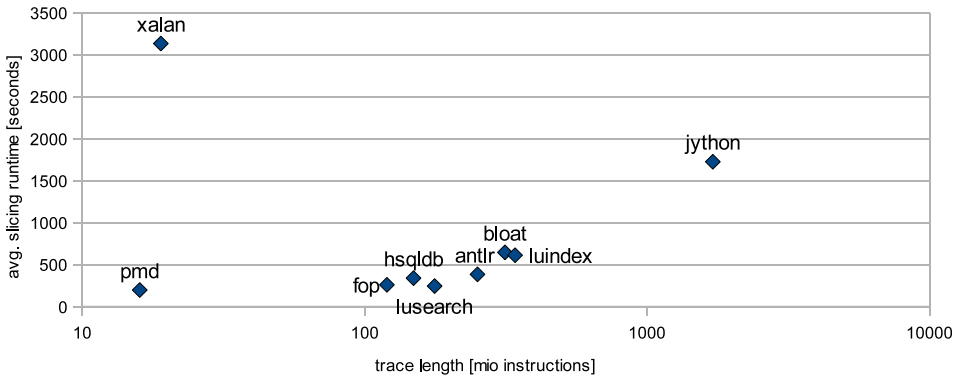

Figure 4: Comparing the trace length to the average slicing execution time shows a weak connection

command line[1]. For computing one or several slices out of this trace, the command is also very intuitive[2].

In JSlice, the situation is more complicated: After accomplishing to install JSlice on the target system, you need to create a text file containing the slicing criterion. Since the format of this file is not defined anywhere, you have to use the Eclipse plugin to create it. This plugin also provides the possibility to perform the slicing, but this failed on all systems that we tried. The JSlice documentation gives a hint how to perform the slicing on command line[3], which results in a text file containing the result.

- *Separation of Tracing and Slicing*:
  In contrast to JSlice, our tool strictly separates the tracing from the slicing component. This allows a developer to create the trace file on one machine, and compute one or several slices on another system. The trace file does not only contain the execution trace, but also a representation of all classes that have been instrumented. Hence the trace file is the only input needed by the slicer.

- *Accuracy*:
  Running JSlice on the source code given in Section 6.1.1 and 6.1.2 shows that it does not dissolve the dependencies as precise as JAVASLICER: In the first example, it additionally points out lines 5 and 7, in the second example the lines 9, 19 and 23. This indicates that JSlice does not correctly resolve the dependencies of the parameters of method calls. Additionally, it always adds the slicing criterion to the slice, which is not always useful.

[1] java -javaagent:tracer.jar=tracefile:my_trace.log,compression:gzip -jar buggy_program.jar

[2] command for slicing all occurrences of line 21 in method `myMethod` of `Class1`:
java -jar slicer.jar my_trace.log 1 Class1.myMethod:21

[3] /usr/local/kaffe/bin/java -noclassgc -slicing -foreclipse _criterions result.log -classpath /usr/local/kaffe/lib/Klasses.jar:/usr/local/kaffe/lib/kjc.jar:buggy_program.jar MainClass

7 Conclusions and Future Work

In Sections 1 and 2 of this thesis, we clearly emphasized the helpfulness of dynamic slicing, especially for automated debugging. The amount of research put on this topic indicates that there is a big interest in dynamic slicing. So far, the main problem was that the only tool available (JSlice) is unusable for most software projects, since it cannot slice most modern Java programs.

We showed that JAVASLICER has the ability to succeed JSlice. Its design makes it easy to integrate it into existing debugging tools. The most interesting question for us is how much dynamic slicing can improve other automated debugging approaches. The evaluation of this question was quite hard so far, since JSlice does not aim at being used by other tools. Using the well-designed API of JAVASLICER, maybe another bachelor's thesis could concentrate on this task.

Our work is just the basic implementation of a dynamic slicer. There are a lot of ways how to extend JAVASLICER to make it even simpler to use or to improve its capabilities. Furthermore, there are various ways how our tool can improve the efficiency of other automated debugging approaches.

Some of our ideas are listed below.

- One way to extend the capabilities as well as the fields of application of JAVASLICER is to make it better trace *multithreaded* programs. So far, we trace each thread separately, which means that we are able to reconstruct the precise execution trace for each single thread.

  For multithreaded programs, it would be most interesting to reconstruct the data dependencies between threads, as well as the precise interleaving of instructions executed by arbitrary threads. To trace these informations is by far not easy. To our knowledge, it has not been shown so far whether is it actually possible to do this using bytecode instrumentation. Of course it would be easy to extend each traced event by a timestamp which is fine granulated enough to distinguish the order of the events, but this would not incorporate caching effects for example. So this approach is not adequate.

- Even if the runtime of the tracer and the compression efficiency of the trace file are already quite impressive, there is still a lot of room for improvements. Using static analysis techniques, the runtime of the tracer as well as the size of the trace file could be further reduced. Furthermore, the serialization of a Sequitur grammar has been implemented quite straightforward. Using advanced techniques like arithmetic coding, the size of the resulting trace file could be decreased by a constant factor of at least 2 or 3.

- In order to make JAVASLICER more easily usable by developers, it would be nice to have an *Eclipse plugin* that allows the selection of the slicing criterion, as well as the slicing itself on any open Eclipse project.

- There are many automated debugging approaches that compare successful program runs with failing runs to estimate for each instruction the probability that it contributes to the erroneous behavior. Some of them use *spectrum based fault localization* techniques [1] for computing these probabilities. Dynamic slicing could also be used to further narrow down the source of the error.

  One concrete work that was lacking a dynamic slicer for Java has been developed at our chair by Kevin Streit [10]. He had to fall back on simpler methods that just use the intersection of the whole execution trace, which had much worse results than using a dynamic slicer.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In Phil McMinn, editor, *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07)*, pages 89–98. IEEE Computer Society, September 2007.

[2] ASM - Home Page. http://asm.objectweb.org/.

[3] Melinda-Carol Ballou. Improving software quality to drive business agility. Technical report, IDC, June 2008. Sponsored by: Coverity Inc.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[5] M. Ducasse. A pragmatic survey of automated debugging. In *Proc. 1st Workshop on Automated and Algorithmic Debugging*, volume 749 of *LNCS*, 1993.

[6] JSlice - a Java Dynamic Slicing Tool. http://jslice.sourceforge.net/.

[7] The Kaffe JVM. http://www.kaffe.org/.

[8] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, Inc., 1999.

[9] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference, 1997. DCC '97. Proceedings*, pages 3–11, 1997.

[10] Kevin Streit. APS - Using automated predicate switching to locate errorprone code regions in java programs, June 2008. Bachelor's Thesis.

[11] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 512–521, 2004.

[12] Tao Wang and Abhik Roychoudhury. Dynamic Slicing on Java Bytecode Traces. *ACM Trans. Program. Lang. Syst.*, 30(2):1–49, 2008.

[13] Baowen Xu, Zhenqiang Chen, and Hongji Yang. Dynamic slicing object-oriented programs for debugging. In *In IEEE International Workshop on Source Code Analysis and Manipulation*, pages 115–122, 2002.

[14] Jianjun Zhao. Dynamic slicing of object-oriented programs. *IPSJ SIG Notes*, 98(38):17–23, 19980515.