View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by KU ScholarWorks

Exploring the Suitability of Existing Tools for Constructing Executable Java Slices

Divya Iyer

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Committee:

Dr. Prasad Kulkarni: Chairperson

Dr. Arvin Agah

Dr. Bo Luo

Date Defended

The Thesis Committee for Divya Iyer certifies
That this is the approved version of the following thesis:

Exploring the Suitability of Existing Tools for Constructing Executable Java Slices

Committee:

Chairperson

Date Approved

Acknowledgement

I would like to thank Dr. Prasad Kulkarni for his help and guidance.

Abstract

Java is a *managed* programming language, and Java programs are executed in a *virtual machine* (VM) environment. Java VMs not only execute the primary application program, but also perform several other *auxiliary* tasks at runtime. Some of these auxiliary tasks, including the program profiling and security checks, are typically performed inline with the program execution, stalling the application progress and slowing program execution. We hypothesize that the execution of individual inline auxiliary tasks do not need knowledge of the entire previous program state. In such cases, it might be possible to abstract individual or sets of auxiliary tasks, along with the code statements that compute the required program state, and execute them in a separate thread. The results of such abstracted auxiliary tasks can then be made available to the main program before they are needed, eliminating the execution time stall and improving runtime program efficiency. However, to test this hypothesis we need access to robust tools to determine and generate the program *slice* required for each auxiliary task.

The goal of this thesis is to study existing Java slicers, test their ability to generate correct *executable* slices, and evaluate their suitablity for this project. Additionally, we also aim to compare the size of the static slice with the minimal dynamic program slice for each task. This comparison will allow us to determine the quality of the static slicing algorithm for Java programs, and provide us with knowledge to enhance the slicing algorithm, is possible. To our knowledge, one of the most robust static Java slicer implementation available publicly is the Indus Java Analysis Framework developed at Kansas State University. We also found the latest dynamic java slicer, which was developed at Saarland University. For this thesis we study these two state-of-the-art Java slicers and evaluate their suitability for our parallelization project. We found that although the Indus slicer is a very efficient and robust tool for the tasks it was originally intended to perform (debugging), the slicer routinely fails to produce correct executable slices. Moreover, the code base has several dependences with older (and now defunct) libraries and also needs all source code to be compiled with obsolete JDK compilers. After a detailed study of the Indus static slicer we conclude that massive changes to the Indus code-base may be necessary to bring it up-to-date and make it suitable for our project. We also found that the Saarland dynamic Java slicer is able to correctly produce dynamic program slices and has few dependences on other tools. Unfortunately, this slicer frequently runs *out of memory* for longer (standard benchmark-grade) programs, and will also need to be updated for our tasks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Future computing environments will more urgently demand high-level programming languages, portable binary distribution formats, and safe and secure runtime systems, in addition to high execution performance. High-level *managed* languages, such as Java and C#, along with their associated virtual machines (VM) have been designed to provide the features demanded by future software systems. VMs provide their features and attempt to improve execution efficiency by performing a number of *auxiliary tasks* at runtime. Some of these tasks, like dynamic compilation and garbage collection, are necessary to ease coding, provide safety, and ensure good performance for the managed environment. Other auxiliary tasks, such as bytecode verification and security management, ensure security of the host platform as the runtime executes untrusted Internet-based applications. Auxiliary tasks that collect profiling information are essential to ensure high program performance while supporting portable execution in a virtual machine. As future systems demand even greater security and performance, it will become necessary for VMs to support increasingly more auxiliary tasks at runtime. However, such auxiliary tasks, performed during execution, have the potential of seriously affecting the final program performance. Moreover, future applications and user requirements are only expected to further increase the number of even more expensive auxiliary tasks to be performed at runtime [5,7,15,24]. Current VM technology is ill-equipped to handle this scenario since it is forced to perform several of these auxiliary tasks *inline* with, and by stalling, normal program execution, thus substantially degrading program performance. Consequently, a critical goal of current VM researchers is to develop techniques to reduce the performance impact of performing expensive auxiliary tasks at runtime [4,14,16,17].

At the same time, the computing community is recently adapting to a major shift in hardware processor design. Instead of scaling uniprocessor clock frequencies, future processors are expected to improve program performance by integrating increasingly more processing cores on a single chip each year [1]. Thus, future program performance is most likely to come from parallelizing and distributing the program workload over multiple cores. This research project attempts to exploit the abundant computing resources on multicore and many-core processors to reduce the overhead of inline auxiliary tasks and improve overall program performance.

1.1 Project Hypothesis

We hypothesize that most auxiliary tasks don't need precise knowledge of the entire program state to do their work, but only the partial state that is relevant to the task in question. Then, we can employ the technique of *program slicing* to statically derive a new program thread (for each auxiliary task or set of tasks) that only calculates the program state necessary for the successful completion of that task. In other works, program slices specialized and optimized for a particular task have been found to be significantly shorter than the main program thread [3, 28]. This characteristic can allow each specialized program slice to run concurrently with the main program thread, and calculate and (if needed) convey the results of its specialized task to the main thread at appropriate points during program execution. Thus, the main program avoids spending time in tasks that are not directly related to the actual execution, allowing unhindered and accelerated program execution. Additionally, since each program slice will itself calculate the required program state, and the main program needs little information from typical profiling or security-related inline auxiliary tasks, communication and synchronization between the main thread and other auxiliary threads are kept to a minimum.

Thus, validation of this hypothesis can allow us to employ a new VM execution model as illustrated in Figure 4.6. At the top of the figure is presented the flow of the program as it exists in current virtual machines, with multiple security and profiling tasks interspersed with normal program execution. The new framework will statically create a separate thread that executes in its own processing core for each such task. The figure illustrates that the specialized slices in each thread are expected to be smaller than the corresponding calculation of the entire program state in the main thread, allowing the auxiliary threads to run ahead and calculate their information before it is needed by the main thread. The main program thread is now free to run without undue suspensions slowing it down. The dotted arrows from *Security thread-1* and *Security thread-2* in figure 4.6 indicate points where information regarding the status of each check may be communicated to the main thread.

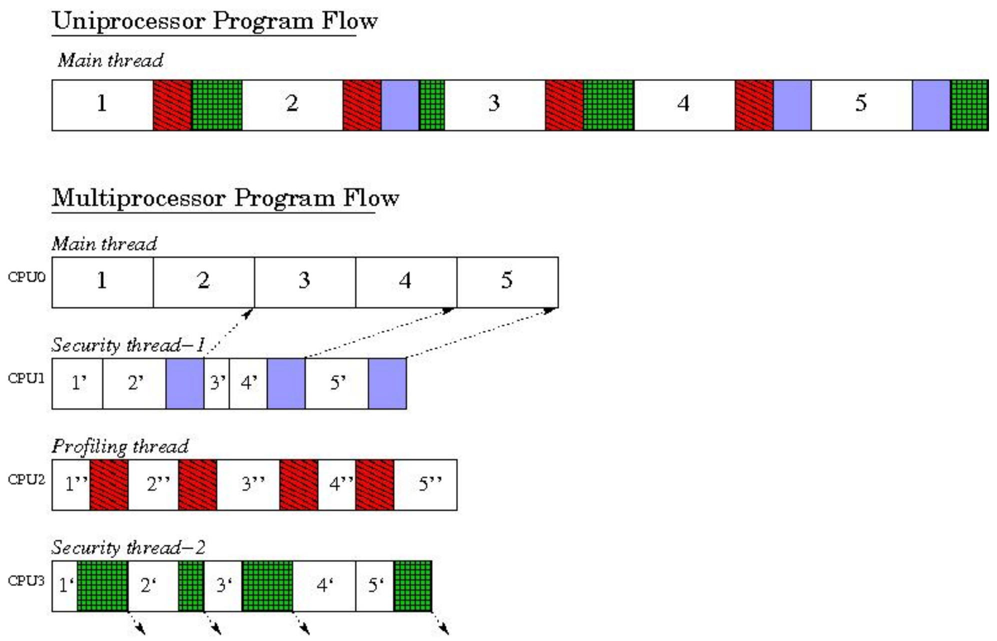


Figure 1.1. Expected Parallelized VM Execution Model

1.2 Goal of this Thesis

The goal of this thesis is to determine the suitability of existing open-source software tools to produce the *executable* program slices required for this research. We are interested in calculating backward slices of a given program. The backward dynamic slice for any given program task consists of the minimum necessary set of program statements for accurate completion of that task. At present, very little tool support exists for slicing programs written in modern object-oriented languages such as Java, and C#. To obtain backward slices for our research we use the state of the art Indus Java Analysis framework [20], developed at Kansas State University. The effectiveness of the static slicing algorithm at producing minimal slices can be quantified by comparing the number of instructions executed by the static slice program with the *dynamic* slice of the same program for the same criteria. We employ the Dynamic Java Slicer [8] developed at Saarland University, Germany to perform this comparison.

Over the last few decades a considerable amount of work has been done in the field of program slicing specific to Java. Unfortunately, there are not many publicly available static slicer implementations which are efficient and robust. The static Java slicer that comes as part of Indus Analysis framework is an open-source and efficient slicer for Java programs. The Indus Java Slicer has the ability to produce forward and backward slices on a given Java program, for the specified criteria. The slice is calculated based on information provided by dependence analyses and also the call-graph provided by Object Flow Analysis. It is claimed that Indus also has the functionality to produce executable slices as class files. This is done by a process called *residualization*. Indus also includes Kaveri [11], a feature rich Eclipse based GUI for slicing. In Indus, Java programs are represented in Jimple via SOOT [23] library. The static slice obtained from Indus can be be further examined by looking at pre and post residualized Jimple files. Theoretically, Indus provides very good functionality for our research.

There has been a lot of work done on dynamic slicing, and such slicers are commonly available for languages like C/C++. Unfortunately, there are very few open-source dynamic slicers for Java programs. Saarland Java Slicer can be used to produce traces of Java program executions, and then offline computing dynamic backward slices on them. This is the only efficient dynamic Java Slicer available for free. Unlike Indus it does not have a GUI interface but is yet quite easy to use. Criteria specification is quite straightforward. The result of slicing gives a count of the number of instructions in the slice along with the bytecode instructions that make up the dynamic slice. Saarland Java Slicer is robust and modern. Its functionality makes it the best possible candidate for our research work.

1.3 Outline

The remainder of this thesis is organized as follows. We start by describing some of the related work in our research area. Chapter 3 talks about the some background concepts and program slicing basics. Chapter 4 gives a detailed explanation of static slicing. We describe our work, observations and results with the Indus static slicer in Chapter 5. We describe the details of dynamic slicing and our research with the Saarland dynamic slicer in Chapters 6 and 7 respectively. Finally, we present the conclusions of this thesis along with the expected future work we plan to conduct in Chapter 8.

Chapter 2

Related Work

Static program slicing, which is a compile-time version of the analysis, was first introduced 1982, whereas run-time based dynamic slicing systems appeared around 1988. Mark Weiser for the first time gave a formal definition of slices and their abstract properties, a practical algorithm for slicing and some experience slicing real programs [26]. Since then a great deal of research has been conducted on static slicing and an excellent survey of many of the proposed techniques and tools can be found in [22] and [10]. [10]presented an evaluation and comparison of implementations of program slicing. His report also discussed how to compare slicing tools for different application areas and formulated a method for doing so. Recognizing the need for accurate slicing, Korel and Laski proposed the idea of dynamic slicing [12]. The data dependences that are exercised during a program execution are captured precisely and saved. Dynamic slices are constructed upon users requests by traversing the captured dynamic dependence information. An exception to this approach is various works on forward computation of dynamic slices [13, 29] which precompute all dynamic slices prior to users requests. [9] proposed an improved slicing algorithm for Java. They presented a new, safe criterion for termination of unfolding nested parameter objects. They also provided measurements for JavaCard benchmark programs. But their work is not finished yet and the slicer would eventually be integrated into the VALSOFT project.

There have been a large number of publications along with a small number of implementations for languages such as FORTRAN, ANSI C, and Oberon. Over the last few decades a considerable amount of work has been done in the field of program slicing specific to Java. Unfortunately, there are not many publicly available static slicer implementations which are efficient and robust. Also most of the research work has been targeted towards particular applications of program slicing such as program comprehension, testing, program verification, etc. There was some considerable static program slicing work(Java program Analyzing Tool - JATO) done at Georgia Institute of Technology in the 90s. But it is mostly irrelevant today. A research group at University of Wisconsin started a Program Slicing project but did not end up with any implementation or experimental work. Many papers claim results form static and dynamic slicing tools but as of January 2011 the only publicly available, reasonably efficient and "working" static slicer for Java is Indus, developed at the Kansas State University [21].

Like static slicing there has been a lot of theoretical work regarding dynamic slicing of object-oriented programs specifically, and ways to store the execution trace in an efficient way. The work by [25] found a great way of storing the execution trace, which is based on Java byte-code. They achieved a much better compression than if they would just write out the sequence of byte-code 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 that considered dynamic slicing especially on object-oriented programs: [27] 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.

Concerning practical implementations of all this theoretical work, it looks very sparsely. To our knowledge, there exists a dynamic slicer for Java, called JSlice. 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. Kaffe is written completely in C, so they changed the source code to write out all information needed to reconstruct a 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 [8]. One has to compile it for every platform, and install it in parallel to the existing Java VM. The source code 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. Apart from JSlice, the only other dynamic java slicer is the dynamic Java-Slicer developed at Saarland University [8].

Chapter 3

Background Concepts

Since its invention by Weiser [26] in the early 1980s, various researchers have explored techniques, variations, and applications of program slicing for over two decades in sequential contexts. Over this period, the primary applications of program slicing were in maintenance, debugging, and testing of programs. Although the application domain of program slicing has been expanded to include program verification and program security, program slicing as a technique is not yet widely used. This is mainly due to the lack of an accessible implementation to experiment with program slicing. There are only a few full-featured robust implementations of program slicing that are available for industrial applications or academic research. Also, in particular, very little tool support exists for slicing programs written in modern object-oriented languages such as Java or C#.

In computer programming, program slicing is the computation of a set of statements from the program that affect the values at some point of interest. The statements that are points of interest are known as the slicing criteria and all statements affecting or affected by the slicing criteria form the program slice.

Calculating static and dynamic slicing involves extensive data dependence analysis and control dependence analysis. 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 these constraints, the static slice is usually relatively big, and not too helpful. Much more precise is the dynamic slice, which is a subset of the static slice and is smaller.

There can be different slices for the same program for different sets of criteria. There is always atleast one slice for any given criteria, which is the program itself. The interesting slices are the ones which, compared to the original program, are significantly smaller and simpler yet, behave like the original program. For our research we are interested in slices that are executable so that we can analyze the performance benefits of running independent program tasks in parallel. We are interested in calculating backward slices of a given program. The backward dynamic slice for any given program task consists of the minimum necessary set of program statements for accurate completion of that task. For example, the dynamic backward slice for a task that prints out a program variable would consist of the program statements which compute the value of that variable upto the point of execution of the printing task.

3.1 Basics

Formally, program slicing is an analysis that accepts a program P along with a collection of program points C and provides a collection of program points S such that either S influences the behavior of P at C or C influences the behavior of P at S [26]. The collection of input program points C are referred to as slice criteria while the collection of output program points S are referred to as a program slice of P . In many literature, the criteria is also accompanied by a collection of variables that determine the subset of the behavior, i.e. the values of the variables, that should be preserved at the criteria in the slice. Below is an example of slicing on a small program [26].

```
int i;
int sum = 0;
int product = 1;
for(i = 0; i < N; ++i) {
    sum = sum + i;
    product = product *i;
}
write(sum);
write(product);
```

This portion of code below is a valid backward slicing of the above program with respect to the criterion (write(sum),sum):

```
int i;
int sum = 0;

for(i = 0; i < N; ++i) {
    sum = sum + i;

}
write(sum);
```

In fact, most static slicing techniques, including Weiser's own technique, will also remove the write(sum) statement. Indeed, at the statement write(sum), the value of sum is not dependent on the statement itself.

Program slices in which the slice influences the criteria is referred to as *backward slices* whereas the slices in which the criteria influences the slice is referred to as *forward slices* . This aspect of the slice is referred as the direction of the slice [2]. Independent of the direction, various analysis explore the relations between the criteria and other programs points to determine which portions of the program to include in the slice. As the definition of a variable influences the use of a variable, data dependence relation needs to be considered during slicing. Similarly, as the execution of (control flow to) a program point can be decided by a decision made at another program point, control dependence relation is also considered during slicing. Further, as the influence may be indirect (e.g. program point e1 may influence the control flow to program point e2 which in turn may influence the data accessed at other program points), the analysis will also need to consider the relation between the slice and several other points in the program. Hence, the slice is expanded till a fixed point is reached.

In [21] the authors describe that mathematically program slicing can be perceived as the transitive closure of the union of a collection of dependence relations . Given this perception, if the dependence relations are represented in a directed graph (i.e. program points as nodes and dependence relations as edges), then the algorithm to calculate a program slice is merely a graph search algorithm for a non-existent node starting from the criteria nodes in the appropriate direction (for forward/backward slices). Such graphs that represent the dependence relations of a program are referred to as program dependence graph (PDG) . Figure 3.1 from [21] represents the PDG for the following code.

```
1   public class Example1{
2      void tempConv(int c) {
3         float f = c * 9 / 5 + 32;
4         if ( f < 100 ) {
5            System.out.println("Good Weather " + f);
6            } else {
7               System.out.println("Hot weather " + c);
8            }
9       }
10  }
```

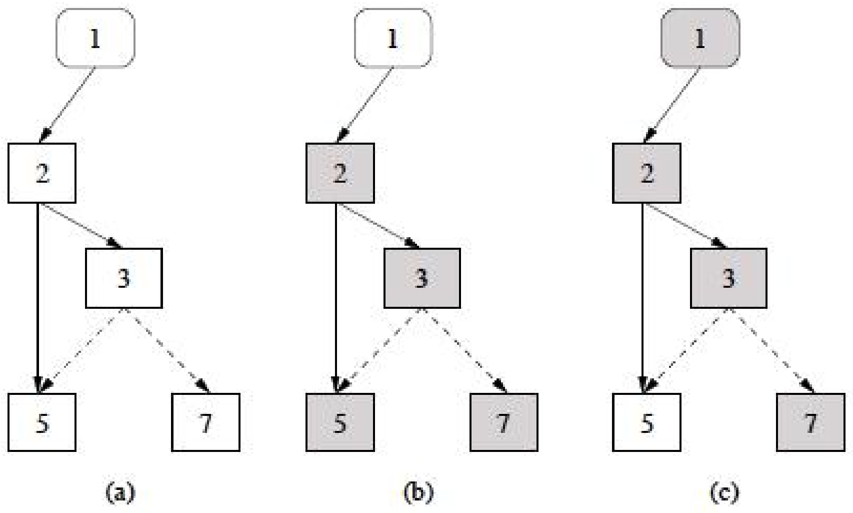


Figure 3.1. (a) : Program Dependence Graph(PDG) for the above example program. Solid lines represent identifier based data dependences while dashed lines represent control dependences.(b) : Forward slice based on criteria line 3.(c) : Backward slice based on criteria line 7.

The basic idea of program slicing is to start with a subset of a program's behavior and reduce it to a minimal form that still produces that behavior. The reduced program is called the *slice*.

3.2 Java Bytecode

One of the main advantages of Java over other programming languages is its platform independence [8]. It is achieved by not compiling the source code into machine code, that would only be executable only on specific systems. Instead, Java uses an intermediate form called bytecode. On the target platform, this bytecode is loaded by the Java Virtual Machine, and eventually translated into machine code that is directly executed on the machine.

The bytecode consists of more than one hundred different bytecode instructions. Bytecode instructions are a level higher than assembly instructions, and almost all of them are translated into several machine code instructions. The Java Virtual machine is stack based and devoid of registers. 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.

When compiled, a .class-file is generated for each single Java class. This class file can be run on any machine that has a Java VM.

3.3 Desirable Slice Characteristics

The properties required of a slice vary from application to application. For example, the program slice required in model checking based program verification applications needs to be executable. On the other hand, executability is not required in applications such as program comprehension via visualization.

There can be different slices, for a given program, for different slicing criteria.

There is always atleast one slice for any given criteria which is the program itself. The interesting slices are the ones which, compared to the original program, are significantly smaller and simpler yet, behave like the original program. The smaller the slice the better but the following argument shows that finding minimal slices is equivalent to solving the halting problem, it is impossible [26].

Definition : *Let C be a slicing criterion on a program P. A slice S of P on C is statement-minimal if no other slice of P on C has fewer statements than S.* *Theorem* : There does not exist an algorithm to find statement-minimal slices of arbitrary programs [26]. Informal Proof: Consider the following program fragment.

```
1   read (X)
2   if (X)
        then
        . . .
        perform any function not involving X here
        . . .
3           X := 1
4   else X := 2 endif
5   write (X)
```

Imagine slicing on the value of x at line 5. An algorithm to find statement-minimal slice would include line 3 if and only if the function before line 3 did halt. Thus such an algorithm could determine if an arbitrary program could halt, which is impossible.

There are two informal properties intuitively desirable in a a slice [26]. First, the slice must have been obtained from the original program after removing the some statements. Second, the behavior of the slice must correspond to the behavior of the original program. A problem with obtaining a slice by removing some statements is that the source code of a program may become ungrammatical. For instance, removing THEN clause from an IF-THEN-ELSE statement leaves an ungrammatical fragment if null statement is not permitted between THEN and ELSE.

```
1        BEGIN
2        READ(X)
3        IF X = 0
4           THEN BEGIN
                        -
              perform infinite loop
               without changing X.
                        -
5                X := 1
6                END
7        ELSE X := 2
8           END
```

Unfortunately the second desirable property is also too strong to be true. In the above example let the slicing criterion be the value of X at line 8. A slicing algorithm based on equivalent behavior for all inputs would necessarily include line 5 unless there were some assurance that for all input, line 5 was never reached. Such a slicing algorithm could be used to determine the termination of an arbitrary procedure by suitably inserting that procedure between lines 4 and 5, and then noticing whether or not line 5 appeared in the slice. But there can be no algorithm to determine if an arbitrary procedure must terminate, and hence no such slicer exists. This is similar to the halting problem discussed previously. The desirability of equivalent projected behaviors of the slice and program can be weakened to be : projected behaviors must be equivalent whenever the original program terminates. So, there is no need to determine arbitrary procedure terminations in the program.

3.3.1 Applications of Program Slicing

Program slicing is most often used to do one of the following:

3.3.1.1 Debugging

When debugging software, it is often the case that a bug is detected at a state associated with single program point Pb. If the software is large and complex, then it is likely that the software fault occurs at a program point Pf that is statically distant from the program point Pb. In such cases, the developer will need to methodically sift through the source code to identify the faulting program point Pf . To expedite this process, the developer will try to limit the search to those parts of the software that directly or indirectly affect the behavior (state) of the program at the program point Pb. This process can be automated using backwards program slicing starting with Pb as the slicing criteria.

3.3.1.2 Program comprehension

Software developers are frequently assigned to debug, further develop, or reverse engineer code that they did not author. In such cases, it is often difficult for the developer to grasp the basic architecture and relationships between code units, and this is made more difficult by the fact that the code may be poorly documented. Both backward and forward slicing can be applied to browse the code, looking for dependences between code units, flows of data between program statements, etc.

3.3.1.3 Testing

There are a number of applications of slicing in the context of testing. One particular example is impact analysis, which aims to determine the set of program statements or test cases that are impacted by a change in the program, requirements, or tests. For example, in verification and validation efforts on large code bases with huge test suites, it is often very expensive to run all the tests associated with the program. If a program statement Pb is modified (e.g., due to a bug fix), rather than re-running all tests, backwards slicing using Pb as the criteria can be used to determine the subset of the tests that actually influence the behavior of the program at the point of the bug fix, and only those relevant tests need to be re-run. In addition, a developer may want to understand the potential impact that the change at Pb can have on other statements of the program. Forward slicing with Pb as the criteria can be used to locate other statements within the program that will be impacted by the change at Pb.

3.4 Dependence Relations

3.4.1 Graphs

In connection with program slicing two kinds of graph are used: flow graphs and dependence graphs.

3.4.1.1 Flow Graph

A flow graph is a directed graph where the vertices represent program statements. They can be used to represent the control and/or data flow of a program. Control-flow graphs (CFGs) are of particular interest in program slicing [11]. A CFG is a flow graph with one unique entry vertex and one unique exit vertex. It models the flow of control in a program so that each path from the entry vertex to the exit vertex represents a possible run of the program.

3.4.1.2 Dependence Graph

We have previously mentioned that the graphs that represent the dependence relations of a program are referred to as program dependence graph (PDG) and how a PDG can be used while slicing.In a PDG the statements and control predicates in a program are represented by vertices and the data and control dependences are represented by edges.

A data dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the order of the two statements is changed. A control dependence exists between a statement and a control predicate if the value of the predicate controls the execution of the statement. The edges from a predicateare labelled with the conditions for which there is a dependence.

A *dynamic dependence graph* is a dependence graph where every executed statement occurs once for every time it is executed. The parts of a program that are not executed are therefore not in the graph [19]. Statements that are executed many times are on the other hand represented by many vertices. The PDG can be augmented with other kinds of edges to represent other dependences. For example, to represent a program with procedures we need special kinds of edges to represent the inter-procedural dependences. A *system dependence graph* (SDG) is an extension of a PDG that have special edges to represent these dependences.

3.4.2 Dependences

Dependence can be thought of as a relation between two program points x and y that indicates if x depends on y or vice versa. In a dependence relation between x and y where x depends on y, we refer to x as the dependent and y as the dependee.

Data dependence can indicate if the variable being read at a program point is influenced by another program point at which the same variable is being written. Similarly, control dependence indicates if the flow of control to a program point is dependent on another program point. To relate a CFG with the program that it represents, there should be a function to map a CFG node n to the code for the program statement that corresponds to that node. Node *n* is control-dependent on *m* in program *p* if there exists a path from *m* to *n* in *p*'s CFG [19].

Data dependence [19] is a notion that relates program points based on the data defined and used at those points.

```
   public class Example1{
    void tempConv(int c){
    float f = c*9/5+32;
     if ( f < 100)
      System.out.println(‘‘Good Weather’’ + f);
}else {
      System.out.println(‘‘Hot Weather’’+ c);
    }
9          c = 20;
10         }
11  }
```

Trivial example to illustrate dependences~\cite{Ranganath_anoverview}.

The variable c is *defined* upon entering the procedure tempConv(int). The definition of c at procedure entry will be used in the definition of f at line 3. In other words, the use of c at line 3 is dependent on the definition of c at procedure entry. Similarly, the use of f at line 3, 4, and 5 is dependent on the definition of c at line 2. However, none of the uses of the variable f depends on the definition of c at line 7 due to the lack of a control flow path.

Chapter 4

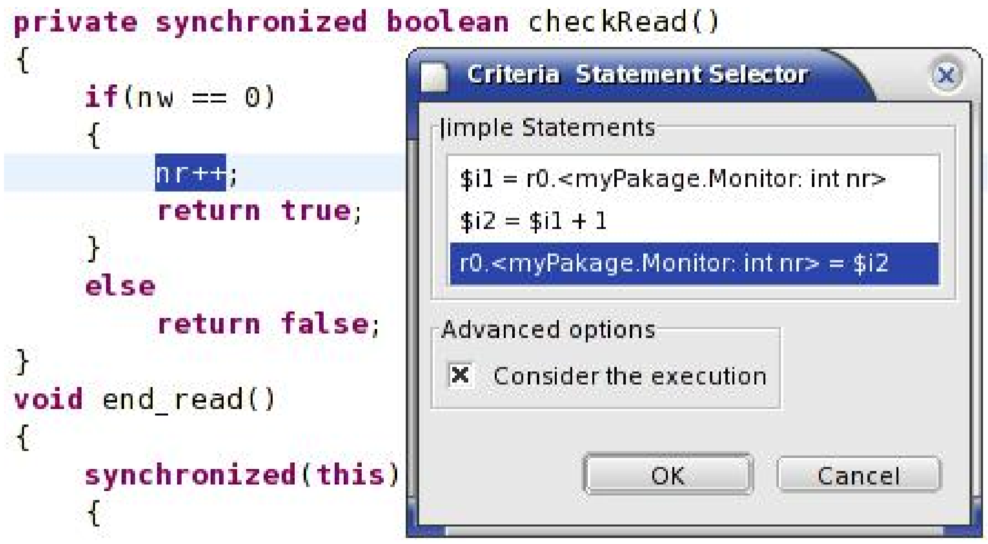
Static Slicing and Indus

The notion of slicing where the algorithm starts at the criterion nodes and looks backward through the program's control- flow graph to find other program statements that influence the execution at the criterion nodes is know as *backward static slicing* [18].

4.1 Indus Java Program Slicer

As mentioned previously, a lot of good background research has gone into the development of the Indus framework. The framework comes with a Static Analyses tool which can be used separately from the slicer. The slicer is built on top this analyses tool. Using the Indus slicer, one can slice object oriented Java applications very easily. Indus also comes with its own Eclipse plugin called Kaveri. Kaveri can be used to slice a Java application and the resulting slice is viewed as highlight annotations on the source code. Adding a statement as criteria is very straightforward as shown in 4.1.

Kaveri displays the jimple code corresponding to each Java statement in the

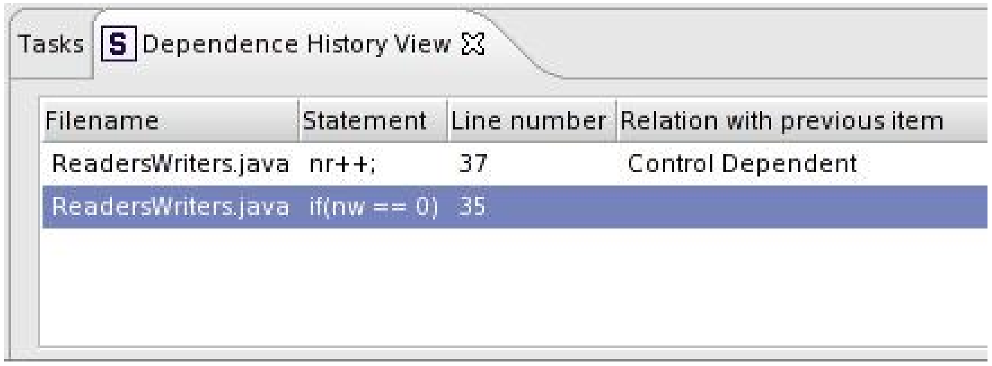


```
private synchronized boolean checkRead()
{
    if(nw == 0)
    {
        nr++;
        return true;
    }
    else
        return false;
}
void end_read()
{
    synchronized(this)
    {
```

```
$i1 = r0.<myPakage.Monitor: int nr>
$i2 = $i1 + 1
r0.<myPakage.Monitor: int nr> = $i2
```

Figure 4.1. Criteria statement in Kaveri [11]

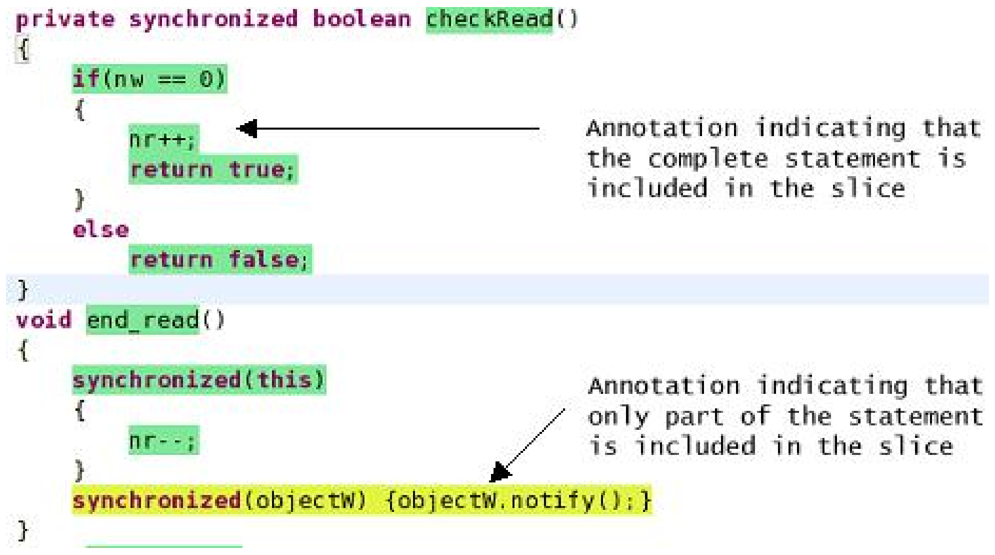
program. As the translation between Java and Jimple is not one – one, the user is allowed to pick the Jimple statements that are closest to the program point of interest. The user can also choose if the effect of executing the criterion be preserved in the slice. When running the slice, the configuration and the criteria can be chosen. Kaveri can also be used to explore a chain of dependencies



Filename	Statement	Line number	Relation with previous item
ReadersWriters.java	nr++;	37	Control Dependent
ReadersWriters.java	if(nw == 0)	35	

Figure 4.2. Dependence History view [11]

using the Dependence History view as as shown in Figure 4.2. The aim being to get a better understanding of the program through dependence tracking. The dependence history view shows the chain of dependences in the form of a stack. The user can backtrack to an earlier point in the dependence chain and follow other dependences. The view indicates the statements that were selected and the dependence that was followed to reach that statement.



```
private synchronized boolean checkRead()
{
    if(nw == 0)
    {
        nr++;
        return true;
    }
    else
        return false;
}
void end_read()
{
    synchronized(this)
    {
        nr--;
    }
    synchronized(objectW) {objectW.notify();}
}
```

Annotation indicating that the complete statement is included in the slice
Annotation indicating that only part of the statement is included in the slice

Figure 4.3. Slicing result seen as highlighted statements [11]

Figure 4.3 shows the slice as highlight annotations on the source text. The green highlight indicates that the statement is completely present in the slice (all the Jimple statements mapped to the corresponding Java statement are included in the slice) while the yellow highlight indicates only a subset of the Jimple statements mapped to the corresponding Java statement are included in the slice.

4.2 Soot and Jimple

Soot [23] is a Java optimization framework. It provides four intermediate representations for analyzing and transforming Java bytecode: 1. Baf: a streamlined representation of bytecode which is simple to manipulate. 2. Jimple: a typed 3-address intermediate representation suitable for optimization. 3. Shimple: an SSA variation of Jimple. 4. Grimp: an aggregated version of Jimple suitable for decompilation and code inspection. Soot can be used as a stand alone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java bytecode.

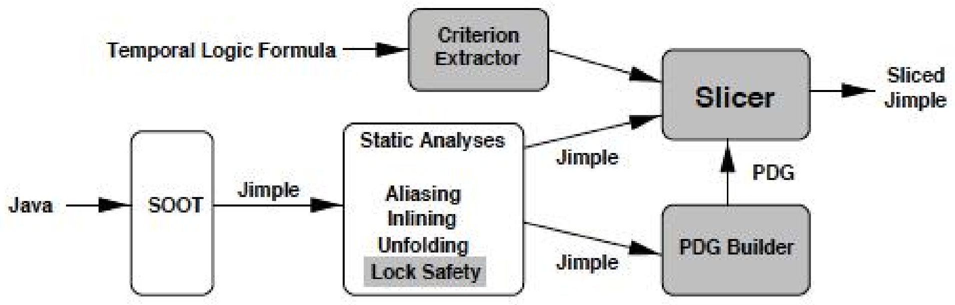


Figure 4.4. Indus's internal transformations [6]

As shown in Figure 4.4 Indus uses Soot to convert Java source code into an internal representation called Jimple. Jimple is the principal representation in Soot. The Jimple representation is a typed, 3-address, statement based intermediate representation. Jimple representations can be created directly in Soot based on Java source code(up to and including Java 1.4) and Java bytecode/Java class files(up to and including Java 5).

The Jimple code is cleaned for redundant code like unused variables or assignments. An important step in the transformation to Jimple is the linearization (and naming) of expressions so statements only reference at most 3 local variables or constants. In Jimple an analysis only has to handle the 15 statements in the Jimple representation compared to the more than 200 possible instructions in Java bytecode [23]. Below is an example from [23] to demonstrate java to jimple translation.

```
public class Foo {

        public static void main(String[] args) {

                Foo f = new Foo();

                int a = 7;

                int b = 14;

                int x = (f.bar(21) + a) * b;

             }

        public int bar(int n) { return n + 42; }

}
```

Running Soot using the command *java Soot.main -f J Foo* yields the file Foo.jimple in the storage directory of soot.

```
public static void main(java.lang.String[]) {

        java.lang.String[] r0;

        Foo $r1, r2;

        int i0, i1, i2, $i3, $i4;

        r0 := @parameter0: java.lang.String[];

        $r1 = new Foo;

        specialinvoke $r1.<Foo: void <init>()>();

        r2 = $r1;

        i0 = 7;

        i1 = 14;

        // InvokeStmt

        $i3 = virtualinvoke r2.<Foo: int bar()>(21);

        $i4 = $i3 + i0;

        i2 = $i4 * i1;
        return;
}
public int bar() {
        Foo r0;
        int i0, $i1;
        r0 := @this: Foo; // IdentityStmt
        i0 := @parameter0: int; // IdentityStmt
        $i1 = i0 + 21; // AssignStmt
        return \$i1; // ReturnStmt
}
```

In the code fragment above we see the Jimple code generated for the main and bar methods. Jimple is a hybrid between Java source code and Java byte code. The local variables in Jimple which start with a \$ sign represent stack positions and not local variables in the original program whereas those without \$ represent real local variables e.g. i0 in the main method corresponds to *a* in the Java source.

Jimple is not Java source, especially the introduction of new unique variables can result in great difference between result and expectations when you compare the Java source code to the produced Jimple code.

4.3 Backward Slicing Algorithm

4.3.1 Program Points

Program points represent expressions [21]. Specifically, the dependee program points correspond to defining assignment expression/statement and the dependent (use) program points correspond to the atomic expression that contains the use variable or procedure invocation expressions.

The same representation suffices for control dependences (CD) as well except for the special handling of two corner cases: 1) the program point of an if statement is the program point of the controlling expression, and 2) the program point of a synchronized statement is the program point of the lock expression of the statement.

4.3.2 Parametric Slicing Algorithm (PSA)

The algorithm [21] accepts a program along with the slice criteria and the set of dependence relations to be consider during the slice construction. This initial set of slice criteria is referred to as *seed slice criteria.* In other words, the algorithm calculates the slice at the fixed point of the function involving the slice criteria, starting from the seed slice criteria and an empty slice. The slice generated by the algorithm is controlled by four parameters:

seedCritGenerator accepts a slice criteria and returns a slice criteria. It functions as a transformer that can add, delete, and/or modify the input slice criteria.

DepHandler processes the given criterion against the given set of dependence relations and provides a new slice criteria to consider. The algorithm includes the program points in the returned criteria into the slice, if they previously did not occur in the slice.

ProcAscHandler provides the program points in the caller (calling) procedure that should be processed to capture the influence/dependence on the given program point.

4.3.3 Backward Slicing Algorithm (BSA)

The parametric slicing algorithm is customized to generate backward slice in Indus [21] by providing appropriate parameter functions. The functions are explained below.

Backward-SeedCritGenerator Returns the seed slice criteria without any modification.

Backward-DepHandler Identifies the dependee program points that influence the criteria program points according to the given dependence relation and returns the dependee program points as new slice criteria.

Backward-procAscHandler Identifies the caller side program points that transfer the control to and influence the data at the program points in the given criteria and returns these caller side program points as new criteria to be processed. Given a program point in a procedure, the execution of the program point depends on the invocation of the procedure.

Backward-procDscHandler Identifies the callee side program points that transfer the control out of the callee procedures. Given an invocation program point, the completion of execution of this program point depends on the completion of the exit program points in the called procedure.

The accuracy of the backward slice depends on the accuracy of the call graph information and the dependence relations used by the algorithm.

4.4 Design Rationale

As shown in Figure 4.5 In Indus dependence information and program slicing is not be tightly coupled as the latter depends on the former while the former does not depend on the latter [11]. Hence, the slicer is separated from dependence analyses. The slicer module depends on a dependence module to provide the information it requires via a well defined interface. The generation of slice and ensuring any property, such as executability, required of the slice, are part of a post processing phase and are kept disjoint from the slicing engine.

Most literature on slicing do not make the distinction between the identification of the slice and the representation of the slice as they do not consider the end application. The reason being that, by definition a program slice is the just some parts of the program picked based on some algorithm by tracking dependences and this process only concerns the identification of these parts and nothing more. The application that uses the slice decides on the representation of the slice. For example, if the application is a visual program understanding tool, it may require the slice to be represented as tagged AST nodes. This indicates that the process of identification of the slice and the representation of the slice are two different activities and the design is further modularized by breaking down the post processing into slice post processing phase and residualization phase.

4.5 Indus Architecture

As described in [11] the slicer is available as a single unit with many modules. Each module is assigned a particular functionality. Each module also provides a well defined interface if to enable for extension by the user. Based on this design principle, the following modules exist in the slicer:

slicer This module is responsible for the identification of the slice, hence, it contains classes required to generate slice criteria, classes that contain the algorithm to identify the slice, and classes to collect the slice. The slice is identified by annotating the AST nodes that are part of the slice.

slicer.processing This module contains various forms of post processing that can be performed on the identified slice in the slice post processing phase. For example, the functionality of making a sliced executable is realized as a class with a well-defined interface.

slicer.transformations This module contains classes that transform the program based on the identified slice.

tools.slicer This module contains classes that package all the relevant parts required for slicing as a "slicer" tool that can be readily used by the end application.

toolkits This module contains adapter classes that adapt classes available in tools.slicer module to be amenable to a tool kit via preferably a tool API.

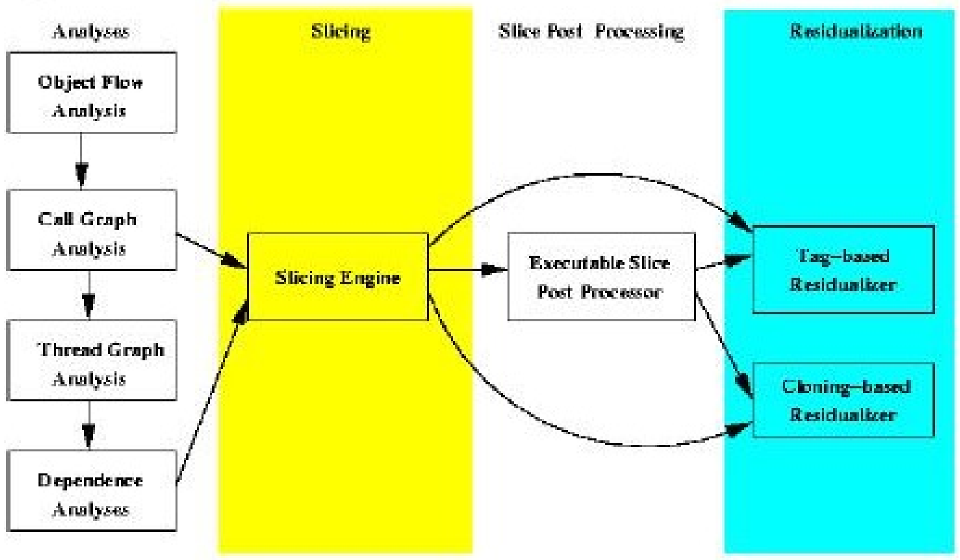


Figure 4.5. Indus Design [11]

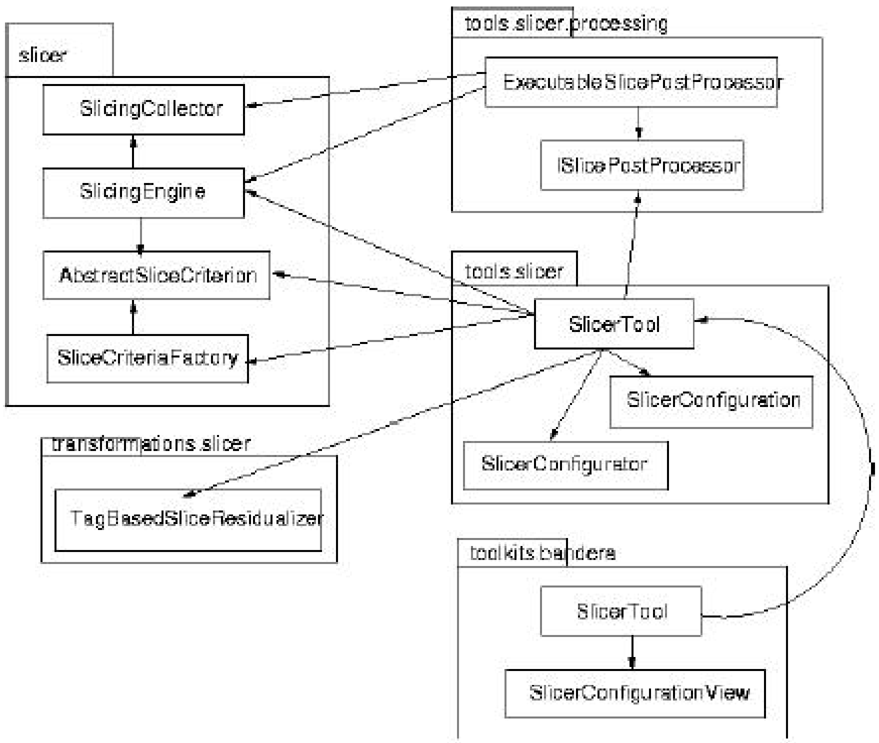


Figure 4.6. UML-style dependence/relationship between various modules in the slicer [11]

4.5.1 Slicer Implementation Details

tools.slicer.SliceXMLizerCLI is a class that uses the tools.slicer.SlicerTool to generate the slice and it residualizes the slice as an XML document and each class in the slice as a Jimple file and a class file. Following is a snippet of the main code in this class [11].

```
public static void main(final String[] args)
{
        final SliceXMLizer _driver = .....
        _driver.initialize();                       (1)
        _driver.execute();                       (2)
        _driver.writeXML();                       (3)
        _driver.residualize();
}
protected final void execute()
{
        slicer.setTagName(nameOfSliceTag);        (4)
        slicer.setSystem(scene);                  (5)
        slicer.setRootMethods(rootMethods);       (6)
        slicer.setCriteria(Collections.LIST);     (7)
        slicer.run(Phase.STARTING_PHASE, true);   (8)
}
```

(1) The XMLizer is created and initialized in the first 2 lines of main. This is followed by the execution of the slicer which (2) is followed by the writing of the slice and the substrate program as XML, writeXML() and the residualization of the slice (3) as Jimple files and class files at residualize().

(4) An annotation-based approach is used to identify the slice. The inherent support in Soot to tag AST nodes is used to identify the slice, hence, in this step we provide the name of the tag that should be used to annotate AST nodes of the program to identify them as belonging to the slice.

(5) Soot uses a "Scene" as an abstraction of the system that is being operated on. All the classes and it's components can be accessed from the Scene via well defined interfaces. To use the slicer the user loads up the classes that form the system into a Scene and provides it to the slice in this step.

(6) Given just the criteria, the slicer can include parts of the system that may not be relevant in a particular run. Although this information is useful in impact analysis, it is overly imprecise in most cases. Hence, the user could identify the set

of methods in the system that should be considered as entry point while generating the slice. The identified entry point methods or root methods can be provided to the slicer in this step.

(7) The slice criteria is set in this step. As the slicer was designed and implemented as part of a larger model checking project, the SlicerTool has the logic that can be switched on to auto generate criteria which are crucial to detect deadlocks in the system. These criteria would correspond exactly to enter_monitor and exit_monitor statements.

The tools.slicer package comes with a default configuration that is used if none are specified and it controls the toggling of various configuration switches. This default configuration uses all possible dependences in their most precise mode to calculate an executable backward slice that preserves the deadlocking property of the system.

(8) The slicer tool executes in 3 stages: starting/initial, dependence calculation, and slicing. A user just wanting to customize the residualization process can extend SlicerTool to alter the post processing phase suitably and use the extended version. The classes from tools.slicer.processing and transformations.slicer will be used in the post processing phase.

The program slicing library, directly or indirectly, requires various high level analyses such as escape analysis, monitor analysis, safelock analysis, and analyses to calculate and prune various dependences – method-local data dependence, inter-procedural data dependence, control dependence, data interference dependence, ready dependence and synchronization dependence. These high level analyses require low-level analyses and information such as object-flow information, call graph, and thread graph. The implementation of all of these analyses and a few more are available in Indus.

Most of the above mentioned analyses are available as independent modules. Hence, a user can use an analyses independent of other analyses in Indus. Moreover, each implementation is separated from it's interface. This enables the user to experiment with various implementations that provide the same interface. In fact, this feature is used in the slicer to vary the level of precision. This theme applies to other analyses available in Indus [21].

Chapter 5

Using Indus to obtain backward executable slices

As good as the Indus slicer is, it still did not provide us with all the functionality that we needed for our research. It is difficult to express the specific criteria intelligently and accurately. And we also need some debugging information to validate the static backward slices.

5.1 Writing Criteria

Indus is a state of the art static Java slicer. Being the only available static Java slicer it is also quite robust and efficient. The analysis tool within Indus is quite efficient and provides accurate information. The first step when slicing a Java program is choosing appropriate criteria. The desirable characteristics of a slice have already been discussed previously in this thesis. In order to get usable slices we should select correct program tasks as our criteria points. The criteria to Indus slicer can be provided as a simple text file with the class file and line number information. A more precise method of specifying the slice criteria, is by means of an xml file. For a simple Java program, specifying a task to print a variable's value as criteria as a line based criteria would be very straightforward. We just need to write <ClassName>:<LineNumber> pairs in the txt file. We can specify several lines in a program as the criteria in similar Class Name, Line Number pair format. But this provides insufficient information to slicer and results in inaccurate slicing results. Specifying just a print statement for a variable as the criteria, does not indicate to the slicer if we are interested in how the value of the variable was calculated. To achieve this we need to write a more complex xml based criteria file which provided the slicer all the additional information it needs. Writing an xml based criteria is not easy and involves a deep understanding of how the slice would work. The format for specifying the xml criteria looks as shown below:

```
<slicer:criterion slicer:considerExecution="">
        <slicer:className>Compress</slicer:className>
        <slicer:method>
            <slicer:methodName></slicer:methodName>
            <slicer:returnTypeName></slicer:returnTypeName>
            <slicer:parameters>
                <slicer:parameterTypeName></slicer:parameterTypeName>
            </slicer:parameters>
        </slicer:method>
        <slicer:stmt slicer:index="" slicer:considerEntireStmt=""/>
        <slicer:expr/>
</slicer:criterion>
                Criteria specification xsd
```

As seen above the xml criteria consists of several fields and parameters. The *considerExecution* attribute tells the slicer to consider the execution of the criteria statement before calculating the slice. The *considerEntireStmt* tells the slicer to consider the entire chosen criteria for slicing. This attribute comes in handy when we are interested in multiple parameters involved in a programming task. Since for this thesis we are interested in executable slices we modified the slicer to set the considerExecution attribute to true by default.

The stmt index in the above xml criteria is not the Java source line number, but the intermediate Jimple statement index. Jimple files are difficult to read and understand. As mentioned in previous sections, each Java statement would map to two or more corresponding Jimple statements. In order to specify the Jimple statement index we need to know the indices of all the Jimple statements that correspond to the task we are interested in slicing. This means an extra step before doing the actual slicing. The user would need to use Soot to create a Jimple file for the Java program, preserving line number information. This generated Jimple file will then be used to see the Java to Jimple mapping and then specify correct indices in the xml criteria. This entire step is very cumbersome and time consuming. So unless, there is a specific need to not include the entire statement in the criteria, we can just set this attribute to be always true. The user can still use the xml based criteria to override these attributes if need be. This makes the process of criteria selection easier and relatively less prone to error. Also, the user need not know the Slicer implementation details or Java to Jimple mapping information in order to choose appropriate criteria.

In our study of Indus, with line number based criteria generation, Indus/Kaveri does the best effort - consider every Jimple (bytecodes) statement that maps to a given line number as a slicing criterion. But, in the absence of such mapping information, the line based criteria generation will be inaccurate. In such situations, it is better to resort to xml based criteria specification.

5.2 Executable slices

For our research we were interested not only in correct slices but also their executability. Executable slices are created in Indus by the process of "residualization". Residualization produces an executable sliced class file for the original program based on the criteria we specify. Indus/Kaveri is correct in producing these executable slices but we also need to verify if the slice is indeed correct and contains all the necessary information. The only way to do is run the slice and observe if it behaves in the same way as the original program. In order to do this we decided to debug the slice in Eclipse IDE and observe if the program flow for the criteria remains the same as the original run.

In order to successfully debug any Java program, along with the source code information, we also need the line number information to step through the program. Unfortunately, the Java-Jimple-Java translations within Indus are lossy and the resultant slice no longer contains essential class file information. Since this information is very crucial to our research, we incorporate an additional step of adding line numbers to the class file.

The first and obvious approach was to try to modify the slicer such that no important information is lost during Indus's translations. We do this as a two step process. The first step is to ensure that the Java-Jimple translation does not lose any information. Indus uses Soot to read in class files and create intermediate representations. We tweaked the slicer to use Soot in a way such that all line number information is preserved during Java to Jimple translation. In the second step we use the line number information attached to the jimple representation, and add it as Line Number Tags to the resultant slice. For each method in every class of the resultant slice, we iterate through the Jimple and line number information and attach the same to the corresponding class file mapping. This again involves a call to Soot since we perform an internal transformation on the sliced class file. This step of adding important class file information is an optional step in the slicer and unless specified, the slicer would continue to produce slices devoid of line number information.

Once we have the above information we try to debug the slices in Eclipse IDE. We step through the sliced program and observe if it behaves like the original application. For jvm98 benchmark suite's *compress program* the slice demonstrated similar behavior as the original program and was also smaller in size. But slicing other benchmark programs for different criteria did not produce desirable results. In order to explore this problem, it was essential to understand if the dependence analysis in the slicer was indeed correct. We used the Kaveri plugin for Eclipse to do this. We used the dependence tracking view of Kaveri to see the dependence analysis results. Using the dependence analysis view it is easy to see if the slicer would produce a correct backward slice. On examining the analysis being done by Indus, we discovered that the inaccuracy in slicing is due the inaccuracy in resolving dependences due to StringBuffer class of Java. Some inaccuracies also creep in when the application is intensive on Swing/AWT classes. Unfortunately, no simple solution is available for this problem. Trying to change the Dependence Analysis is out of scope for this thesis. It would involve changing quite a bit slicer code which is built on legacy Java and Soot framework.

5.3 Results and Observations

The static slicer used for our experiments is part of the Java Analysis Framework called Indus and is available for free download at http://indus.projects.cis.ksu.edu/. To run the slicer we use JDK 6 along with its JRE. The input to the static slicer is Java class file which has been compiled using JDK4. This a very strict requirement and the slicer would not work for programs that are compiled using any JDK later than JDK4. Indus uses Soot for all its internal translations and transformations. Although Soot is available free to download from its project website, a slightly modified version of Soot-2.1.0 should be downloaded from the Kansas State University's Indus project website. This setup should suffice to run the static slicer.

In order to build our own slicer, some very specific packages including but not limited to groovyMonkey, jibx, slf4j, commonsLogging, etc. are needed. The complete list of packages is available on the Indus web page. The Indus plug-in for Eclipse is part of the Indus build and can be used in Eclipse after installing the Groovy Monkey plug-in. Kaveri plug-in works only on Eclipse 3.1 or earlier. We need ant installed on the computing machine in order to build the Indus project. Any version of ant can be used. Indus slicer is not memory intensive and any amount of memory greater than 512MB should be enough to run standard Java Programs.

We use different flags in the Indus static slicer command line tool to obtain pre and post residualized jimple files and the sliced class files. The slicer supports Linux operating systems. We used Fedora 10 during our research. We use the standard benchmarks from the jvm98 suite to conduct our experiments with Indus.

5.3.1 Results and observations

As mentioned and described in previous sections, Indus's static slicer is the only static slicing tool available. Using Indus for our research means relying on a lot of dependencies, the worst being the requirement that all class files passed as argument should have been compiled with JDK 4. Indus only takes class files as arguments and we cannot give it the Java source file as the input to read. This poses a twofold problem. Firstly, this makes it absolutely necessary to have the source code of the program readily available so that it can be compiled with JDK4. Secondly, compiling newer programs can give rise to compilation problems such as Class or method "not found" or "not defined". This also means using old and deprecated Java APIs. Also, the version of Soot used for the internal transformations is quite old and results in lossy translations sometimes. Indus depends on a version of Soot that can only handle Java 1.4. While there are versions of Soot that can handle Java 1.4+, these versions break the dependences required by Indus, e.g. IIRC, MethodRefs are used instead of SootMethod in InvokeExpr in Soot 2.2 and Indus requires SootMethod in InvokeExpr.

Inspite of the above mentioned problems, we tried to slice the benchmarks in the jvm98 suite through Indus. Since the source code of the benchmark suite is available we compiled the benchmarks using JDK4. And then sliced the benchmarks by using all the program print statements as the criteria. In case of jvm98 benchmarks the print statements display the final value of the variables used at various points in the program and also print out benchmark run information. For example, for compress benchmark the print statements display the size of an input file before and after compression. So when we select these print statements as the criteria, the expected backward slice would contain all the program tasks that contributed to the calculation of the variable values upto the chosen print statements. The only way to verify our theory is to run the sliced(residualized) class see if it behaves similar to the original program run. In other words the slice should also print out the variable values that depict an input file size before and after compression. Unfortunately, the execution behavior of the slice doesn't turn out to be the same as the original program run. Another alternative to verify the slice correctness is to read the post residualized jimple file and verify that it contains all the necessary information. For a large application program it is impossible to read the jimple files manually and this approach is also prone to error. So we opened the slice in Eclipse debugger and step through the slice execution to observe if it follows the same execution pattern as the original program. Here we found out that the slice execution is alright upto a certain point after which it demonstrates buggy behavior. We did the same for other benchmarks in the jvm98 suite and discovered that the slice dependences are not resolved correctly when the source code uses certain StringBuffer Java classes. These inaccuracies in the slice creep in due to inaccurate dependence analysis. Solving this behavior is outside the scope of this thesis.

The goal of the project includes successful generation of static slices which can be run on the available resources. Indus slices the application programs successfully and residualization process generates executable slices. Although these slices are runnable they do not depict correct behaviors and more often than not run into NullPoniterExceptions when executed. This is again because of the missing information and in StringBuffer classes. Since we were not able to get any accurate executable slices, we do not have precise numbers to validate our project premise. Comparing the pre and post residualized jimple files, as well as the original program's and slice's class file information, clearly shows that the slice is definitely smaller than the original program and therefore our hypothesis stands true. But actual performance benefits of slicing can be measured only when we have a slice that is accurate as well as executable.

Chapter 6

Dynamic slicing and Java slicer

Dynamic slicing algorithms can greatly reduce the debugging effort by focusing the attention of the user on a relevant subset of program statements. Recognizing the need for accurate slicing, [12] proposed the idea of dynamic slicing . The data dependences that are exercised during a program execution are captured precisely and saved as traces. Dynamic slices are constructed upon users requests by traversing the captured dynamic dependence information. Precise dynamic slices can be considerably smaller than static slices. While precise dynamic slices can be very useful, it is also very expensive to compute dynamic slices.

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 [8], that are not very useful in practice and can hardly be used in automated debugging to isolate possible error sources.

Once a program is executed and its execution trace collected, precise dynamic slicing typically involves two tasks: *preprocessing* which builds a dependence graph by recovering dynamic dependences from the program's execution trace; and *slicing* which computes slices for given slicing requests by traversing the dynamic dependence graph.

The basic approach to dynamic slicing is to execute the program once and produce an execution trace which is processed to construct dynamic data dependence graph that in turn is traversed to compute dynamic slices. The execution trace captures the complete runtime information of the program's execution that can be used by a dynamic slicing algorithm – in other words, there is sufficient information in the trace to compute precise dynamic slices. The information that the trace holds is the full control flow trace and memory reference trace. Therefore the complete path followed during execution and each point where data is referenced is known. For a dynamic slice to be computed, dynamic dependences that are exercised during the program execution must be identified by processing the trace.

The computation of this trace is the most critical part in dynamic slicing. 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.

6.1 Saarland Dynamic Java Slicer

6.1.0.1 Design

Saarland Dynamic JavaSlicer can be used to produce traces of Java program executions, and then offline computing dynamic backward slices on them. It is organized in several modules [8] :

Tracer : This is the java agent which produces a trace file of a java program execution. The trace file contains the bytecode representations of all loaded classes, as well as all information to reconstruct the execution trace for each thread.

TraceReader : Contains all classes needed to open and process a trace file. It provides forward and backward iterators over the execution trace.

Core : This is the core of the slicing component. It uses the TraceReader component to process a trace file and computes all dynamic dependences inside this run. The slicer itself is built on top of that. It accumulates the dynamic dependences to compute the dynamic backward slice.

6.1.0.2 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.

A Java agent is used to instrument the bytecode of all classes that are currently loaded into the VM, and the new classes as they are loaded. A naive instrumentation of a small Java program increase the method to hundreds of instructions. 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 element in the vector in order to be able to compute a precise slice. The instrumentation of these standard library classes also introduces some problems.

For tracing the control flow of the execution, all possibilities for jumps in the control flow are considered. These also include *exceptions*. Also method invocation sites are considered.

Trace File

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

6.1.0.3 Slicer

For dynamic slicing basically the Saarland slicer uses the algorithm presented in [25]. Additionally Saarland slicer extends 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.

The dynamic slicing algorithm essentially proceeds by traversal. The crucial operation in constructing slices is to detect the control/data dependences between bytecodes. The data dependence analysis is complicated by Java's stack based architecture as explained in the following.

Saarland Dynamic Java slicer, like Indus static slicer, exploits stacks to explore data dependence between program variables during execution. The main challenge that remains with the above algorithm 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, the complete control flow graphs for each method are built in [8], and a mapping that assigns each instruction the set of all instructions depending on this one is stored. This computation takes some time, but since it is only done once per method, this has no negative consequences for the overall efficiency.

Chapter 7

Saarland Dynamic Java Slicer and our reserach

For our research we are interested in the number of instructions in the trace file(both application and library) as well as the application instructions in the trace file. The trace Reader does provide the total number of instructions in the trace file. But this count contains application, library as well as some additional instructions which are introduced in the trace file due to various instrumentations. We extended the trace reader such that it would count only the application instructions in the execution trace. The original functionality of counting all the instructions also remains intact.

Since the trace file is huge, it was impossible to read the trace file of a standard Java program without having to print all the library instructions as well. We added the functionality in the trace reader such that, a user can view only the application instructions and skip printing of the library instructions.

The result of the dynamic slicing gives a count of the number of instructions in the slice along with the bytecode instructions that make up the dynamic slice. We also need to know the total number of instructions in the slice for fair comparisons in this thesis. To achieve this we modify the Slice Instructions Collector in the dynamic slicer to use a map instead of a set and associate a counter with each instruction. Each time an instruction is visited to be put in the slice this counter is updated. At the end of the instruction collection we get the total number of instructions in the slice. Again, the original functionality of the slicer to show the distinct instructions count remains intact.

7.1 Experimental Setup

The requirements to run the dynamic slicer are quite straightforward. The only project requirements are a working JDK on the computing machine. Dynamic Java Slicer works well with the latest JREs and we use JDK6 for our research. The dynamic Java Slicer is also free to download from the project website of Saarland University at http://www.st.cs.uni-saarland.de/javaslicer/.

The Dynamic Java Slicer is built as a maven project with different modules. In order to download and customize the Slicer maven should be installed and running on the system. The dynamic slicer is memory intensive and a large Java program can take upto several hours to finish slicing on a machine with insufficient memory. For the slicer to run smoothly and in a timely manner we provide it upto 4Gb of memory. To test our experiments with the Dynamic slicer we use the standard jvm98 benchmark suite and benchmarks from Dacapo2006 suite.

For our research we used the computing facilities at Information Technology and Telecommunications Center(ITTC), University of Kansas. The benchmarks for dynamic slicing were run on the ITTC clusters with 4 nodes. Complete information about ITTC computing facilities and clusters can be seen at http://bioinfo.ittc.ku.edu/. The dynamic slicer was given 4GB memory to run the jvm98 and Dacapo benchmarks.

7.2 Results and Observations

For running our experiments on the dynamic java slicer we use benchmarks both from jvm98 benchmark suite as well as Dcapao 2006 benchmark suite.

The trace file sizes for the various programs from the above mentioned standard Java benchmark suites is shown in the tables 7.1 and 7.2.

In Table 7.1 we show results of trace files for five jvm98 benchmarks. For the remaining jvm98 benchmarks (mtrt, mpegaudio, jack, raytrace) the tracer ran into errors to the effect of code being compiled incorrectly and byte-code having unsupported form. This behavior doesn't seem too wrong since the Saarland Dynamic slicer is quite modern and the jvm98 benchmark suite might have some outdated Java APIs. We deliberately chose the jvm98 benchmark suite because we wanted to run the slicer for smaller programs before moving on to the comparatively larger and longer ones. We can see that even for very small programs like jess, check and checkit the trace file is several mega bytes in size. Even for compress the trace file size is quite big. For a longer running and large program like db the trace file is several Giga Bytes in size. The third column in Table 7.1 gives the total number of instructions in the dynamic slice - application as well as library. The fourth column gives a count of the total number of application instructions in the slice. Since this calculation reads data from the trace file, it is necessary to keep the trace file in memory till the number of application instructions is calculated. Because the trace file itself is so huge this process runs out of memory before the number of application instructions can be calculated for db.

The time to compute the trace file itself is quite reasonable though.

Benchmark	Size	Total Number of instructions	Number of application instructions	Time to compute trace file
compress	152M	77,139,392	72,344,018	1.06s
jess	23M	8,945,996	4,920,181	0.23s
check	3.3M	524,464	257,546	0.18s
checkit	79M	41,674,309	517,801	0.24s
db	5.6G	2,518,028,036	GC overhead	8.50s

Table 7.1. Trace results for jvm98 benchmark suite

Table 7.2 shows the trace results for eight standard programs from dacapo2006 benchmark suite. The traces for the remaining dacapo2006 benchmarks(jython, eclipse, chart) could not be created because of the huge size of the benchmarks and the tracer ran out of memory. In this case also the time to compute trace itself is not too high. Since the dacapo2006 benchmarks are bigger than jvm98 benchmarks the trace files are several Mega Bytes in size. Even for the smallest program in dacapo2006, pmd, the trace file size is very big. The third and fourth columns show the total number of instructions and the number of applications in the trace file respectively. In this case, only a small fraction of instructions in the trace file are application instructions. The majority of instructions are Java library instructions.

Table 7.3 show the dynamic slicing results for various benchmarks. The criteria for slicing are those program statements which invoke the Java Security Manager for permissions during execution. We can see from the results that the time to compute the slice is not insignificant even when significant resources are given to the dynamic slicer. But the time to compute can be used to gauge the performance benefits since the computation time largely depends on the chosen criteria. Also

Benchmark	Size	Total Number of Instructions	Number of Application Instructions	Time to compute trace file
pmd	165M	27,536,927	762,589	0.27s
fop	183M	113,512,343	254,626	0.50s
antlr	437M	171,019,220	1,922,515	0.44s
hsqldb	411M	272,541,002	13,734,412	0.48s
bloat	618M	358,406,825	105,516,203	0.56s
lusearch	3.0G	192,205,885	15,033,226	2.47s
xalan	2.6G	21,287,268	214,875	2.09s
luindex	912M	383,602,261	5,634,911	1.10s

Table 7.2. Trace results for dacapo2006 benchmark suite

Benchmark	Time to compute slice	Number of Distinct Instructions	Total Number of Instructions	Slice to Program Ratio
compress	26763.56s	1557	25,370,062	0.331
jess	394.76s	2009	1,838,955	0.214
checkit	1005.03s	542	4,636,090	0.106
pmd	557.45s	4429	1,286,513	0.047
fop	498.22s	3466	440,984	0.006
hsqldb	2175.10s	4580	848,805	0.003
antlr	2684.04s	4127	640,167	0.004
bloat	3789.94	3294	841,337	0.002

Table 7.3. Dynamic slicing results for security check instructions as criteria

the slice computation time will depend on how many objects a program allocates during its run and heap space utilization. The count of distinct instructions is obviously quite low. Each instruction is executed several times during the run of the program and this constitutes the slice. The fourth column gives the size of the resulting slice in terms of the number of instructions in it. Although this number is quite high, it is very low compared to the size of the original program. The fifth column gives the ratio of the original program size to the size of the slice. A ratio of less than 1 simply means that the slice is smaller than the program. Not surprisingly, the ratios are very small. Infact, the ratio is smaller for the bigger programs. This clearly validates our thesis goal - verify that the dynamic slice is smaller than the original program.

Chapter 8

Conclusion and Future Work

Indus static slicer[1], though quite robust and efficient, doesn't quite suffice to get all the necessary results needed for validation of our project premise. We modified the static slicer to be able to use simple line based criteria to produce accurate results. Indus's residualized executable slices are devoid of important line number information, essential for debugging the resulting slice. So we also incorporated additional functionality in the slicer to generate "residualized" class files with line number information. Unfortunately, the residualized slices, although smaller than the original program, do not produce similar output. To explore this problem, we extensively examined the dependence analysis results of Indus using Kaveri's dependence tracking view. Inspite of the analysis being quite correct, we discovered inaccuracy in slicing due to inaccuracy in dependences involving String-Buffer classes. Rectifying these dependence inaccuracies is outside the scope of this thesis, and could break several project dependencies. It would also involve considerable changes to the slicer which is built on outdated Java and Soot framework.

The underlying analysis framework within Indus is in reasonably good condition. In the future, a new static slicer could be built on top of the existing analysis framework. Indus could be made to use a modern version of Soot for its internal translations. This would probably also get rid of discrepancies due to incorrect analysis of StringBuffer classes and would also allow modern Java programs to be statically sliced successfully. Another essential goal should be to reduce the strict dependence on outdated jar files and classes. The only other alternative to improving Indus would be to build a static slicer from scratch.

Dynamic Java Slicer [8] involves creation of a trace file for the run of the original program and then running the slicer on this trace file for a set of criteria. The dynamic slicer produces results that validate our project premise. The dynamic slicer's result only depicted the number of distinct application instructions in the slice. We modified the slicer to also depict the count of total number of instructions in the slice. We also modified the trace reader to get the count of total application instructions in the original run of the program, so that we could compare the slicing results against the number of instructions in the original program.

Although the dynamic slicing results are promising, the slicer runs out of memory for larger Java programs. This is a significant drawback and needs to be rectified. A possible solution could be break the trace file into blocks and only that block is kept in memory which is relevant to the current calculation of the slice.

References

[1] International technology roadmap for semiconductors. accessed from http://www.itrs.net/Links/2008ITRS/Home2008.htm, 2008.

[2] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[3] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2):8, 2007.

[4] B. Calder, C. Krintz, and U. Hölzle. Reducing transfer delay using java class file splitting and prefetching. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 276–291, New York, NY, USA, 1999. ACM.

[5] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. *Computer Security Applications Conference, Annual*, 0:463–475, 2007.

[6] M. B. Dwyer, J. C. Corbett, J. Hatcliff, S. Sokolowski, and H. Zheng. Slicing multi-threaded java programs: A case study. Technical report, 1999.

[7] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 108–123, New York, NY, USA, 1995. ACM.

[8] C. Hammacher. Design and implementation of an efficient dynamic slicer for Java. Bachelor's Thesis, November 2008.

[9] C. Hammer and G. Snelting. An improved slicer for java. In *ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22, Washington, DC, June 2004.

[10] T. Hoffner. Evaluation and comparison of program slicing tools. Technical report, 1995.

[11] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering the indus java program slicer to eclipse. *Lecture Notes in Computer Science*, 3442:269â272, 2005.

[12] B. Korel and J. W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[13] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 66–79, New York, NY, USA, 1994. ACM.

[14] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software – Practice and Experience*, 31(8):717–738, 2001.

[15] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: an on-access anti-virus file system. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 73–88, Berkeley, CA, USA, 2004. USENIX Association.

[16] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 198–208, Washington, DC, USA, 2007. IEEE Computer Society.

[17] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 143–154, New York, NY, USA, 2000. ACM.

[18] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29, August 2007.

[19] V. P. Ranganath and J. Hatcliff. An overview of the indus framework for analysis and slicing of concurrent java software (keynote talk â extended abstract).

[20] V. P. Ranganath and J. Hatcliff. Slicing concurrent java programs using indus and kaveri. *Int. J. Softw. Tools Technol. Transf.*, 9(5):489–504, 2007.

[21] tesh prasad ranganath. *scalable and accurate approaches for program dependence analysis, slicing, and verification of concurrent object oriented programs.* PhD thesis, Kansas State University, March 2006.

[22] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 13–23. IBM Press, 1999.

[24] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–169, Washington, DC, USA, 2001. IEEE Computer Society.

[25] T. Wang and A. Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Trans. Program. Lang. Syst.*, 30:10:1–10:49, March 2008.

[26] M. Weiser. Program slicing, 1981.

[27] J. Zhao. Dynamic slicing of object-oriented programs, 1998.

[28] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. *SIGARCH Comput. Archit. News*, 28(2):172–181, 2000.

[29] Ãrpâ¡d BeszÃ©des, T. Gergely, Z. M. Szabo, J. Csirik, and T. Gyimothy. Dynamic slicing method for maintenance of large c programs. In *5TH EUROPEAN*

CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR, pages 105–113, 2001.