Executables from Program Slices for Java Programs

Jason M. Gevargizian

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. Andy Gill

Dr. Perry Alexander

Date Defended

The Thesis Committee for Jason M. Gevargizian certifies
That this is the approved version of the following thesis:

Executables from Program Slices for Java Programs

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Acknowledgements

I thank my advisor, Professor Prasad Kulkarni, for mentoring me throughout my research projects. He often goes above and beyond the call of professorship and without his guidance none of this would have been possible.

I thank all of the other professors who have taught and advised me during my time at the University of Kansas. I would especially like to thank Professors Perry Alexander and Andy Gill, who, along with my labmates, have made me feel welcome in the Computer Systems Design Laboratory and, with whom, I have enjoyed cross-interest collaboration.

I thank my close friend, Zachary Hoffman, who has encouraged me throughout the years through inspiration and friendly competition in my computer science career.

Finally, I would like to thank my family for their love and support; they mean the world to me.

Thank you all.

Abstract

Program slicing is a popular program decomposition and analysis technique that extracts only those program statements that are relevant to particular points of interest. Executable slices are program slices that are independently executable and that correctly compute the values in the slicing criteria. Executable slices can be used during debugging and to improve program performance through parallelization of partially overlapping slices.

While program slicing and the construction of executable slicers has been studied in the past, there are few acceptable executable slicers available, even for popular languages such as Java. In this work, we provide an extension to the T. J. Watson Libraries for Analysis (WALA), an open-source Java application static analysis suite, to generate fully executable slices.

We analyze the problem of executable slice generation in the context of the capabilities provided and algorithms used by the WALA library. We then employ this understanding to augment the existing WALA static SSA slicer to efficiently track non-SSA datapendence, and couple this component with our exectuable slicer backend. We evaluate our slicer extension and find that it produces accurate exectuable slices for all programs that fall within the limitations of the WALA SSA slicer itself. Our extension to generate executable program slices facilitates one of the requirements of our larger project for a Java application automatic partitioner and parallelizer.

Contents

List of Figures

Chapter 1

Introduction

Program slicing is a popular program decomposition and analysis technique that extracts only those program statements that are relevant to particular points of interest [5, 21, 22, 26]. A program slice is the set of instructions that the specific criteria is dependent upon. Program slicing is a useful technique and has found application in program debugging [1, 8, 10, 13], program parallelization [24], program differencing and integration [16, 17], software maintenance [14], testing [3, 15], reverse engineering [4, 9], and compiler tuning [19].

Program slicing algorithms and the slices they generate can be categorized in many ways. One useful categorization is between executable and non-executable slices. An executable slice is program slice that is independently executable and that correctly computes the values in the slicing criteria. Simple program slicers often do not generate exectuble slices and can ignore many key elements that are required by the langauge definition. The control and data dependence relationships between instructions do not express all invoke dependency relationships, which carry with them many language specific considerations. Thus, slicers generating executable slices need to do additional work to find and maintain such depen-

dence relationships in the final slice, as well as include only the other statements that are necessary to generate a syntactically and semantically valid program.

Executable program slices are useful to us to test the correctness of the slicer by ensuring the computed state in the slice is equivalent to the original programs computed state at the point of the slice criteria. Executable slices are useful in isolating behaviours of a program when all behaviors are not desired. Executable slices can provide improved program performance through program parallelization with partially overlapping slices. Executable slices are necessary for the software maintenance techniques proposed by Gallagher and Lyle and the regression testing reduction algorithm proposed by Binkley [5]. A (section of) the program may interleave the computation of multiple independent tasks, which may later all be needed to realize some program action. For example, computing the partial program state to initiate a slow security check, like invoking a remote certificate authority (CA), may be done in parallel to running the remaining program tasks that occur earlier or are independent of the security check. With the use of slicing, programs can be partitioned to allow such partially overlapping program states to be computed in parallel, thus freeing many instructions from costly delays and speeding-up overall program execution.

A simple program slice can be augmented to be executable by tracking invoke dependencies and enforcing other language specific rules. As such, the nature of the problem will vary by language. In the case of Java, work must be done to include information from the exception table, which is a mapping for potential branches not expressed by the program statements (and thus not present in a simple slice). Also, slices are often computed and expressed in an intermediate form that may not exactly match the initial source language nor the final binary

representation. In such cases, a mapping criteria must be established to return to the original form. Furthermore, if the intermediate form is a lossy form (as is the case with WALA's SSA form), the unrepresented data must be tracked alongside the intermediate form to produce a complete and accurate slice. Java also has other rules, like the need for invokes to constructors and superconstructors that often do not have simple data or control dependence over any of the instructions in the simple slice.

WALA, Watson Libraries for Analysis, contains a static SSA slicer designed primarily for analysis of Java programs. WALA uses an SSA, single static assignment, intermediate representation to generate a simplified system dependence graph, which represents program statments along with both control and data dependence. This SSA slicer was designed and works extremely well for some types of analysis but does not generate an executable slice. Furthermore, this SSA form does not represent local variable and stack information present in Java bytecode.

In this thesis, we build a system that extends the WALA Slicer to produce a fully executable slice. This extension accurately tracks the additional non-SSA data dependences and produces a fully executable slice from the SSA-only slice. We have produced this system and have used it to test the performance of the framework.

This thesis is organized as follows. Chapter 2 will present related work, discussing program slicing in more depth and then covering the WALA features that were used most heavily in the proposed framework. In Chapter 3, we will discuss the methodology of the extended slicer by discussing the phases of the slicer's execution and simultaneously showing an example Java program being processed from Java source all the way to an executable slice. In Chapter 4, we will discuss

the future work and how we intend to use our frameworks for program parallelization by program partitioning. In Chapter 5, we present our conclusions.

Chapter 2

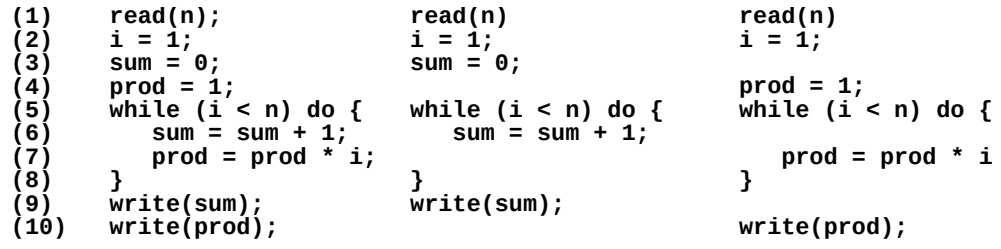
Related Work

2.1 Technology Background

2.1.1 Program Slicing

Program slicing is a popular program decomposition technique that extracts only those program statements that are relevant to particular points of interest [5, 21,22,26]. These points of interest are called *slicing criteria*, and each are typically comprised of the location of a program statement p and a set of program variables v which are in scope at p. As such, a slice is defined as the subset of the original program statements that includes all of the instructions upon which the states of variables in v at statement p is dependent, for all slicing criteria. Figures 2.1 (b) and (c) show examples of static executable slices for two different slicing criteria for the example program in Figure 2.1(a). Figure 2.1(b) shows a program slice with respect to the criteria (9, sum), and 2.1(c) shows the slice for the criteria (10, prod).

Program slicing was first proposed by Mark Wieser as a conceptual abstraction people implicitly use during program debugging [22,23,25], and recommended ex-



(a) Example program

(b) program slice for criteria (9, sum)

(c) program slice for criteria (10, prod)

Figure 2.1. Examples of static program slices

plicit tools to aid the process. Program slicing for debugging was further explored by several researchers [1, 8, 10, 13], and interactive slicer tools for program understanding and debugging are now available [18,20]. Program slicing techniques have since found numerous other applications is areas as varied as program parallelization [24], program differencing and integration [16, 17], software maintenance [14], testing [3, 15], reverse engineering [4, 9], and compiler tuning [19].

Program slices can be derived in a variety of ways. One of the most important distinctions is between a static and a dynamic slice. A static slice is computed without making assumptions regarding a program's input, whereas the dynamic slice depends on some specific test case [2, 27]. Our proposed parallelization model uses an *executable* static program slice, as first proposed by Wieser [22]. Executable slices are supersets of their non-executable counterparts in that an executable slice may contain additional program statements that are required to maintain proper language syntax or to otherwise ensure that the slice can be executed independent of the main program. Both the program slices in Figures 2.1 (b) and (c) are examples of executable static slices.

Detailed empirical studies on the size of program slices reveal that for a precise slicer and without optimizations, the average program slice contains under

one-third of the program [6, 7]. While slices for smaller programs tend to include most program statements, the slices for larger programs have been observed to be considerably smaller than the original program. At the same time, previous investigations show that compiler optimization techniques can improve the accuracy of slices and reduce their size further [11, 12]. Researchers have also reported that sizes for highly optimized and accurate *speculative* slices for instructions causing branch mispredictions and cache misses were less than 10% of the program's dynamic instruction stream for a window of 512 closest instructions [28]. Generating minimal and precise program slices forms a core component of our proposed research as is be described in Chapter 3.

2.1.2 Executable Program Slicers

Executable program slicers produce independently executable slices; id est, complete programs. Executable slices are supersets of their non-executable counterparts in that an executable slice may contain additional program statements that are required to maintain proper language syntax or to otherwise ensure that the slice can be executed independent of the main program.

Grammatech's CodeSurfer is an commercial C static slicer that can produce executable slices.

Indus is a collection of program analysis and transformation tools for Java, part of which is an executable Java program slicer. The Indus Java Program Slicer was implemented as part of a project to work with Bandera, a tool set for model checking of concorruent Java applications. Indus is a static slicer that produces executable slices. We chose not to use this slicer for our work as it does not support slicing of Java programs more recent than those compiled in JDK4.

2.1.3 Static vs. Dynamic Slicers

A static slice is computed through static analysis and thus produces a slice that is correct for any arbitrary input. A dynamic slice is computed from a single dynamic trace and thus is a correct slice for only one set of program inputs.

Static slicers produce static slices; a static slicer will typically compute full control and data dependence graphs to then attempt to find reachability to program statements from the specified criteria to arrive at the slice. Dynamic slicers produce dynamic slice; a dynamic slicer generates a dynamic trace and then works backward through dependencies from the slice criteria to calculate the dynamic slice.

WALA is a Java static analysis suite and has implemented a static SSA slicer, see Section 2.2 for details. Saarland University's JavaSlicer and Wang & Roychoudhury's JSlice are example of dynamic slicers for Java.

For our application of automatic program partitioning and subsequent execution for arbitrary program input (more details in Chapter 5), we required a static slicer such as WALA.

2.2 WALA

WALA is the T.J. Watson Libraries for Analysis. WALA provides many static analysis and manipulation capabilites that our executable slicer makes use of, including but not limited to, a Java bytecode instrumentor, SSA system dependence graph builder, and SSA slicer.

2.2.1 Shrike IR and Shrike Instrumentor

Shrike is an intermediate representation that is, with few exceptions, a 1-to-1 mapping of Java bytecode instructions. For all program analysis in WALA, Java program bytecode is first loaded in and translated into collections of Shrike statements, by class and method.

WALA has a Java bytecode instrumentor that allows loading Java programs, performing manipulations, and then outputing a new Java executable. This is implemented as the Shrike instrumentor, which makes modifications to the Shrike intermediate form. This functionality is utilized by the executable slicer to output the final executable slice. After the executable slice set is determined the original Jar is manipulated using the shrike instrumentor to match the slice.

2.2.2 SSA, the SDG, and the SDG Builder

For the WALA slicer and other analysis, WALA can generate a System Dependence Graph (SDG). The SDG is a graph that represents both a program's data and control dependence for its instructions. The SDG nodes represent individual WALA SSA IR instructions and the edges between the nodes represent control and data dependence. The SDG that it generates is in a single static assignment form (SSA) and thus does not keep track of the java stack and local map instructions. The SDG node are SSA statements, and thus are not 1-to-1 mappings with the Shrike code nor the Java bytecode. Single static assignment form treates all possible states of variables as a new constant and this SSA form, specifically, does not express the local map data nor the virtual stack interaction represented in the Java bytecode. As such, many instructions are not in the SDG; e.g. the push instructions, pop instructions, dup instructions, load instructions,

and store instructions.

The WALA SSA form is sufficient for various types of analysis like WALA's SSA slice, but it is not sufficient to produce a fully executable program. Getting from the SSA SDG and SSA slice to a fully executable slice is described in Chapter 3.

The SDG builder works as a non-deterministic simulator of the input Java program, which is processed in the Shrike intermediate form. As the simulation runs, the SDG is built with state meta-data that is tracked along the way. As statements are visisted for the first time, SSA versions of the statements are created as nodes in the SDG, if there is an SSA translation (others are omitted).

The simulator is non-deterministic in the sense that it takes all branches rather than just one actual execution path. During the similuation, the system states are tracked in a similar fashion to the Java runtime. In the Java runtime environment, a single state maintained and is comprised of primarily a stack and a localmap; the stack is a stack of values and the local map is value mapped to a given local number. In the SDG build simulator, multiple states are maintained for all paths the execution can take and each state is comprised of also a stack and a local map. For the SDG simulator, the stack and local map contain references to an abstract constant. Each constant has a created-by relationship with the node that originally made it.

i	Instruction	Creates	Uses
1	iconst_0	1	
2	store_0		1
3	iconst_5	2	
4	load_0		1
5	addi	3	1, 2
6	store_1		3

Table 2.1. WALA constant tracking.

For example, a binary op (addi) instruction creates a new constant (the result) and uses two existing constants (the operands). See Table 2.1.

As the simulation runs, the SDG nodes are built for the instructions that are visited. Data dependence edges are made when a constant is used; the edge is between the node that used it and the node that created it.

As control statements are visited, scopes are maintained and control dependence edges are added to the SDG.

2.2.3 SSA Slicer

The SSA slicer uses the SSA SDG and user specified criteria instructions to compute the SSA slice. This done by determining reachability from the slice criteria node to all other nodes; nodes that are reachable have either data or control dependence and are therefore part of the slice.

Again, this slice is not sufficient for an executable program as it does not contain information about the stack and local-map instructions.

Chapter 3

Methodology

The goal of this thesis is to extend the WALA SSA Slicer and dependent tools to produce a fully executable slice. The WALA SDG builder has been extended to track the non-SSA data dependence between instructions. A new executable slice backend has been designed to take an SSA Slice, using the non-SSA data dependence, and compute a complete executable slice. The Shrike isntrumentor is then used to manipulate the original Java bytecode to produce an executable Jar from the resulting slice.

Non-SSA data dependence speaks to any dependence not captured in the SSA form. The SSA form does not express the Java virtual stack, thus push-pop relationships are non-SSA. The SSA form does not have local variables, thus load-store relationships are non-SSA.

Some other patches/fixes have been employed to deal with special cases and limitations of the WALA SDG build and WALA SSA slicer that cause issues for the generation of an exectuable slice. These fixes are described in Section 3.4.2.

This Chapter will discuss the methodology of the phases of the executable slicing system in execution order from Java source input to executable slice output. A

running example, input Java program with slice criteria, will be used to illustrate the process. The phases, and organization of this chapter, are as follows:

1. Slice problem preparation and breakup of the multi-branch merge points
2. SDG Build and tracking of non-SSA Data Dependence
3. Single Static Assignment Backwards Slice
4. Complete Slice Generation
   (a) Recovery of non-SSA instructions
   (b) Special cases handling (Exception Table and Exception Handlers, Unused Invoke Pops, Constructors for News, Recovery Phases and Fixed Point)
5. Executable Generation

We use a simple example program to illustrate and explain our approach. The initial source for our running example is shown in Listing 3.1 and the corresponding Java bytecode is shown in Listing 3.2.

Listing 3.1 Running Example's Java Source

```
public class Sample {
        public static void main(String[] args) {
                int x = 0;
                int y = 1;
                int z = 2;
                if (z == 3) {
                        x = 4;
                }
                z = foo(x);
                System.out.println(z);
        }

        public static int foo(int v) {
                System.out.println(v);
                v++;
```

```
                return v;
        }

}
```

Listing 3.2 Running Example's Java Bytecode

```
public class Sample extends java.lang.Object{
public Sample();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   iconst_0
   1:   istore_1
   2:   iconst_1
   3:   istore_2
   4:   iconst_2
   5:   istore_3
   6:   iload_3
   7:   iconst_3
   8:   if_icmpne       13
   11:  iconst_4
   12:  istore_1
   13:  iload_1
   14:  invokestatic    #2; //Method foo:(I)I
   17:  istore_3
   18:  getstatic       #3; //Field
    java/lang/System.out:Ljava/io/PrintStream;
   21:  iload_3
   22:  invokevirtual   #4; //Method java/io/PrintStream.println:(I)V
   25:  return

public static int foo(int);
  Code:
```

```
   0:   getstatic       #3; //Field
   java/lang/System.out:Ljava/io/PrintStream;
   3:   iload_0
   4:   invokevirtual   #4; //Method java/io/PrintStream.println:(I)V
   7:   iload_0
   8:   iconst_1
   9:   iadd
   10:  istore_0
   11:  iload_0
   12:  ireturn

}
```

3.1 Breakup of Multi-branch Merge Points

First, the original jar is modified to separate multi-branch merge points so that phi-nodes remain distinct.

Multi-branch merge points are points in execution where two or more execution paths merge; in other words, it is an instruction that is the target of at least two branches. The number of possible incoming paths merged at a point in the program is the number of branches that target the specified point plus one, for fall through from the previous instruction.

The WALA SSA system dependence graph builder automatically symplifies the phi-nodes of multi-branch merge points into single, mathematically equivalent, phi nodes. This simplification is useful for some types of analysis but does not allow the WALA Slicer to differentiate the control dependence of one branch vs another that share the same target.

EXAMPLE:

The following example shows how the PHI node simplication can result in an incorrect slice. Algorithm 1 shows the source program. Algorithm 2 and Figure 3.1 shows the source with the simplified Phi nodes. Algorithm 3 and Figure 3.2 shows the souce with the divided Phi nodes.

Algorithm 1 Source

```
1: x ← 0
2: y ← 0
3: if c1 then
4:     x ← 1
5:     y ← 1
6:     if c2 then
7:         y ← 2
8: print(x)
```

Algorithm 2 SSA with Simplified Phi Node

```
1: x_0 ← 0        //in slice
2: y_0 ← 0
3: if c1 then     //in slice
4:     x_1 ← 1    //in slice
5:     y_1 ← 1
6:     if c2 then     //in slice
7:         y_2 ← 2
8: y_3 ← φ(y_0, y_1, y_2)
9: x_2 ← φ(x_0, x_1, x_1)     //in slice
10: print(x_2)     //slice criteria
```

The WALA system dependence graph builder simulates the execution of the application and keeps track of all paths and their representative states. When a point in execution is visited that has already been visited, this is considered a *meet*, and a phi node is created or extended for the representative states. States from previous visites are merged, via the phi node, with the current state.

Because the phi node for X in the simplified example is created with information from both the path taken flow for the branch at 6 and the not-taken flow, the

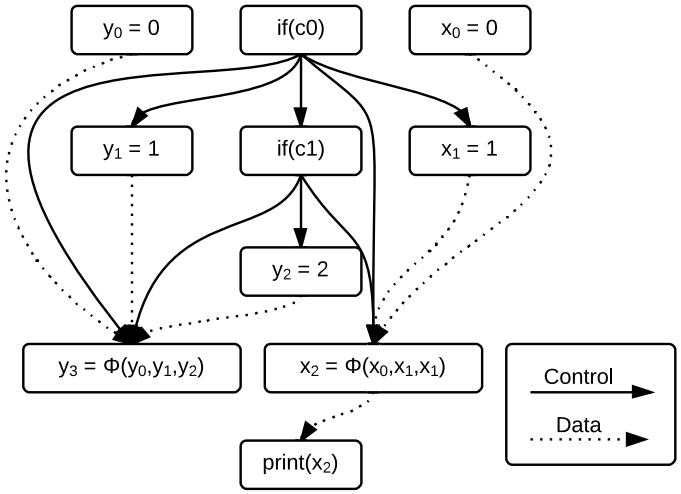


Figure 3.1. Simplified phi node call graph.

Algorithm 3 SSA with divided phi nodes

1: $x_0 \leftarrow 0$ //in slice
2: $y_0 \leftarrow 0$
3: **if** $c1$ **then** //in slice
4: $x_1 \leftarrow 1$ //in slice
5: $y_1 \leftarrow 1$
6: **if** $c2$ **then**
7: $y_2 \leftarrow 2$
8: $y_3 \leftarrow \phi(y_1, y_2)$
9: *nop*
10: $y_4 \leftarrow \phi(y_0, y_3)$
11: $x_2 \leftarrow \phi(x_0, x_1)$ //in slice
12: $print(x_2)$ //in slice

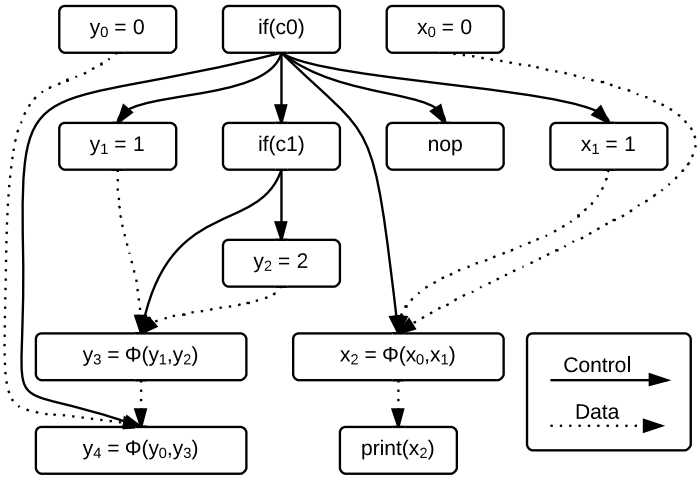


Figure 3.2. Divided phi node call graph.

SDG will be built with control dependence edges from both if-statements. Consequently, the WALA slicer will include both branch statements in the slice because of this control dependence releationship.

In the divided example, a nop instruction is inserted at the merge point to make the targets of the two if-statements different; the inner if-statement now targets the nop while the outer if-branch targets the print(). Consequently, the WALA SDG builder does not combine the meets for the two branches; the SDG can be seen with separate phi nodes of only two incoming states in Figure 3.2. Now, the x variable has control dependence only with the outer if-statement thus allowing the slicer to successfully include the outer if-statment and omit the inner.

We found that a trivial modification of the WALA SDG builder to simply not merge the states was not an option. This phi node merging is necessary in many

cases; for example, loops that meet on the same point multiple times often need to have updated versons of their SSA variables added to the phi node. Without additional static data (like the nop separators), there is no way for the SDG build simulator to differentiate between states that come from the same or different branches.

3.2 SDG Build and non-SSA Data Dependence

The new divided jar is passed along to the WALA SDG builder which produces the SDG. At this point the user specified class exclusions and main class specification is necessary.

The class exclusions define a set of classes to be ignored in the call graph build. This was used by our tests, in Chapter 4, to slice on only the application libraries as opposed to the application libraries and the Java libraries. The main class specification is required to give the SDG builder a start point to build the SDG. WALA's SDG builder is described in more detail in Section 2.2.2

The SDG builder has been extended to track non-SSA data dependence for the non-SSA instruction recovery for the final executable slice. A table, *sourcetable*, has been implemented to keep track of push-pop and load-store relationships between instructions. Hook-ins have been inserted into the SDG build to update the source table as the SDG simulator executes.

In sync with the SDG simulator states, the sourcetable keeps track of simulation states as well. Like an actual JVM runtime state and the WALA simulator state, the non-SSA data states contain a local map and a stack. Instead of the actual values (like the JVM), or the constant values (WALA simulator), the source instruction index is maintained.

For Example; Table 3.1 shows the tracking of the non-SSA data dependence after bytecode instructions executed sequentially.

i	Instruction	Localmap after	Stack after	PushForPop	StoreForLoad
i_1	iconst_0	{}	$\$, i_1$	{}	{}
i_2	store_0	$\{l_0 = i_1\}$	\$	$\{i_1\}$	{}
i_3	iconst_5	$\{l_0 = i_1\}$	$\$, i_3$	{}	{}
i_4	load_0	$\{l_0 = i_1\}$	$\$, i_3, i_4$	{}	$\{i_3\}$
i_5	addi	$\{l_0 = i_1\}$	$\$, i_5$	$\{i_4, i_3\}$	{}
i_6	store_1	$\{l_0 = i_1, l_1 = i_5\}$	\$	$\{i_5\}$	{}

Table 3.1. Sample non-SSA tracking and non-SSA Data Dependence Table.

.

We can see that the stack holds reference to the instructions that pushed the would-be value and the localmap contains references to the instructions that would have stored the would-be value. For each instruction, PushForPop and StoreForLoad mappings are tracked. These mappings are used to recover the non-SSA instruction in the final slice recovery. For example, if i_6 is in the slice, we know i_5 must be included as well since there is a Push-Pop relationship between the two. In Figure 3.3, the running example's resulting SDG is shown. In Table 3.2, the source table is shown with the non-SSA data dependence.

-

-

-

-

-

-

-

-

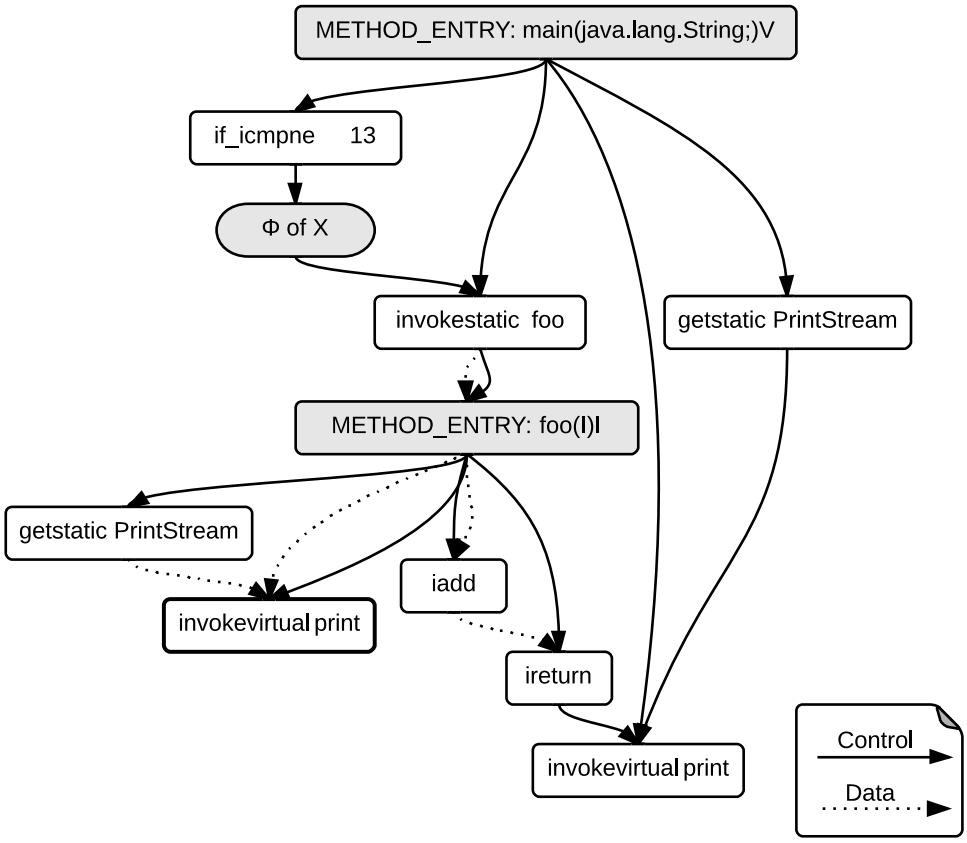


Figure 3.3. Running example's SDG.

-

-

i	Instruction	PushForPop	StoreForLoad
	main		
0	iconst_0		
1	istore_1	{0}	
2	iconst_1		
3	istore_2	{2}	
4	iconst_2		
5	istore_3	{4}	
6	iload_3		{5}
7	iconst_3		
8	if_icmpne...	{6, 7}	
11	iconst_4		
12	istore_1	{11}	
13	iload_1		{1, 12}
14	invokestatic...	{13}	
17	istore_3	{14}	
18	getstatic...		
21	iload_3		{17}
22	invokevirtual...	{18, 21}	
25	return		
	foo		
0	getstatic...		
3	iload_0		p_0
4	invokevirtual...	0, 3	
7	iload_0		p_0
8	iconst_1		
9	iadd	7, 8	
10	istore_0	9	
11	iload_0		10
12	ireturn		

Table 3.2. Running example's non-SSA Data Dependence Table.

3.3 Single Static Assignment Backwards Slice

The WALA Slicer uses the SDG to generate the SSA slice. The SSA slice is a collection of SSA statments that the criteria instruction(s) are depedant upon. The WALA SSA Slicer is described in Subsection 2.2.3.

Figure 3.4 shows the resulting SSA slice for the running example.

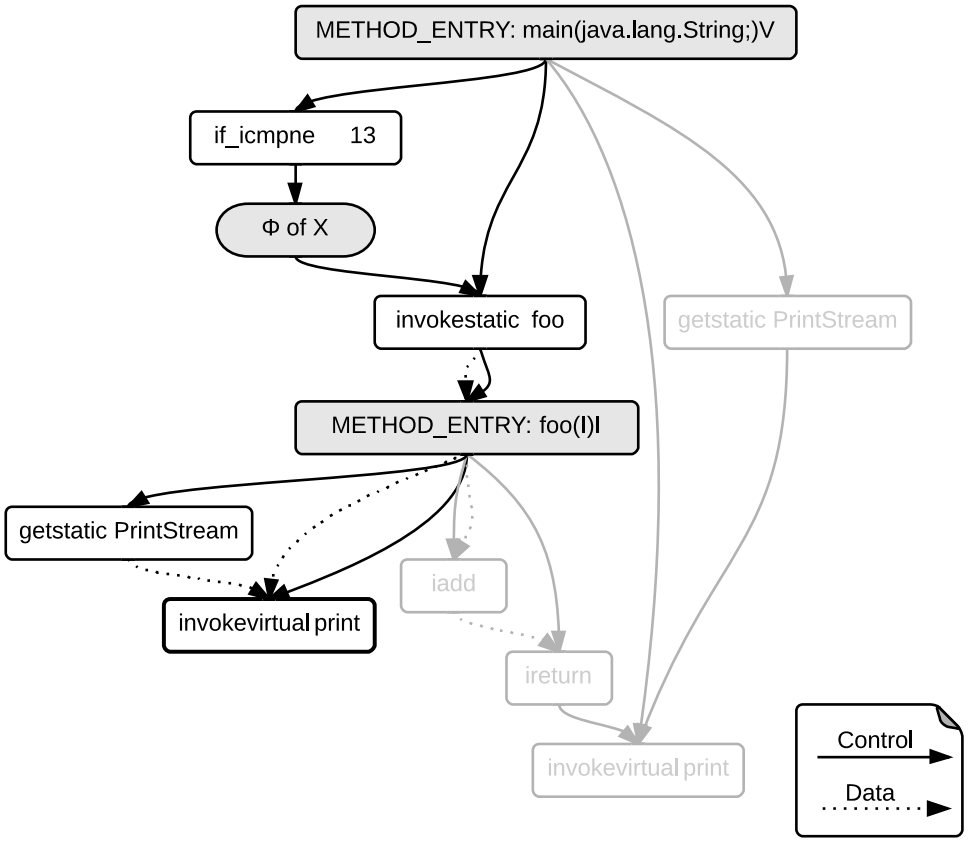


Figure 3.4. Running example's SSA Slice.

3.4 Generate Complete Slice / Recover Stack & Local Instructions

The SSA slice is not sufficient to generate an executable. First, the Java instructions that relate to the Java virtual stack and local map must be recovered.

3.4.1 Main Recovery

The sourcetable is used to recover the non-SSA data dependent instructions. These non-SSA data dependencies are the stack push-pop relationships and the local load-store relationships.

For every instruction in the SSA slice, the Java bytecode equivalent is included in the final slice. Then, for each of these instructions the sourcetable is used to find and include any instructions that have push-pop or load-store relationships with the given instruction. Upon the newly included instructions, the same process is repeated until all non-SSA data dependencies have been exhausted (and included in the final slice). The results are stored the final slice map, a mapping from each instruction to a bytecode-manipulation operation (*include*, *omit*, *replacewithpop*) to be carried out during executable generation.

Figure 3.5 shows the final slice and the non-SSA data dependence as edges. Table 3.3 shows the final slice for the running example.

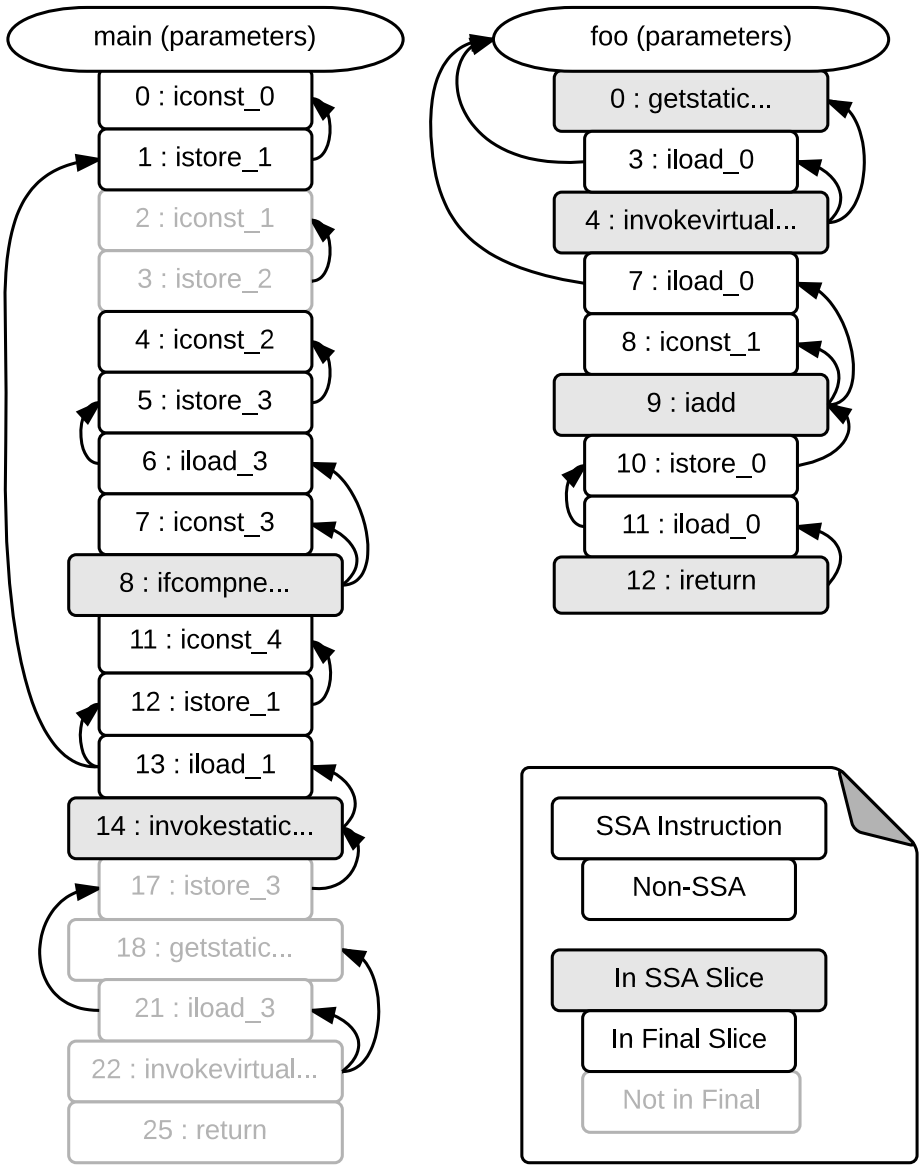


Figure 3.5. Running example's Slice Recovery.

i	Instruction	Operation
	main	
0	iconst_0	Include
1	istore_1	Include
2	iconst_1	Omit
3	istore_2	Omit
4	iconst_2	Include
5	istore_3	Include
6	iload_3	Include
7	iconst_3	Include
8	if_icmpne...	Include
11	iconst_4	Include
12	istore_1	Include
13	iload_1	Include
14	invokestatic...	Include
17	istore_3	ReplaceWithPop
18	getstatic...	Omit
21	iload_3	Omit
22	invokevirtual...	Omit
25	return	Include
	foo	
0	getstatic...	Include
3	iload_0	Include
4	invokevirtual...	Include
7	iload_0	Omit
8	iconst_1	Omit
9	iadd	Omit
10	istore_0	Omit
11	iload_0	ReplaceWithPush
12	ireturn	Include

Table 3.3. Running example's Final Executable Slice.

3.4.2 Special Considerations and Fixes

There are special considerations for a few cases that the main recovery process does not cover to make sure the Java rules are adhered to for actual execution.

3.4.2.1 Dup Instructions

DupInstructions are instructions that duplicate items on the stack. This is essentially a pop of the stack and multiple pushes of the value. There are also some variations of dup that involve duplicating items other than the top element of the stack.

For the executable slicer and non-SSA recovery, it is possible to have a dup instruction included in the slice without the pops for both duplicates in the slice. In this case the dup instruction is removed as it is not utilized in the slice.

i	Instruction	Before Fix	After Fix
1	iconst_1	X	X
2	dup	X	-
3	istore_0	-	-
4	istore_1	X	X

Consider the above example where we have a push followed by a dup followed by two pops. In this case the instruction at 4 (istore_1) is already in the slice. From the pop at 4 (istore_1) we recover the push at 2 (dup) and from the pop at 2 (dup) we recover the push at 1 (iconst_1). Now we have the dup[2] in the slice, which has the stack signature [value $\rightarrow$ value value], but only one corresponding pop is included. Since foo()[3] is not to be part of the slice we can simply remove the dup[2].

3.4.2.2 Recover Super Constructors

The WALA SDG build does not create dependence relationships for object constructors and their super constructors unless the super constructor explicitly initializes variables that are used later in the slice. This suffices for some forms of analysis but not execution of Java code because Java requires superconstructors to be called all the way back to the object class.

For all constructors that are included in the slice, corresponding super constructors must be recovered. This recovery phase looks at all init functions that are included in the slice and finds the superconstructors within them to be included (init invokes within inits).

3.4.2.3 Exception Handlers

The WALA SDG build does not observe the control dependencies between instructions and their exception handlers because non-executable slicers do not need them. To build an executable slice, all instructions in the slice that throw excpetions need to have handlers.

For all exception handlers in the exception table, this phase determines the set of exception handlers that handle exceptions for instructions that are in the slice. For these exceptions, an exception handler is required for the generated executable. The original exception handler is checked to see if it is included in the slice. If the exception handler is not already in the slice itself (because of side effects), this recovery phase replaces it with a generic exception handler.

The generic exception handler is a sequence of two instructions: 1) a goto for the non-exceptional path to jump over the handler, and 2) a pop for the exception handler object.

It should be noted that since the slice is defined as the set of instructions the criteria instructions are dependant upon, exception handlers are only necessary in so far as the are required by the Java code verification process to execute. Unless, the criteria is directly dependent upon or within the exception handler itself; this is generally not the case.

3.4.2.4 Unused Invoke Pops

There are cases where invokes to methods that return a value will be included in the slice because of some side effect but the return value itself will not be needed. In this case, the invoke will be included in the slice but the corresponding pop will not. This violates the stack rules for Java execution as some other instruction will pop this element undesirably.

These instructions needed to be handled by adding generic pop instructions that throw away unused return values. This recovery phase identifies invoke pushes that do not have corresponding pops and adds generic pop instructions after the invoke.

3.4.2.5 Constructors for the 'New' Instruction

The WALA SDG build does not create a dependence relationship between the NEW instruction and the invoke of init for said object unless there are elements in the init that are later used in the slice. This works for some types of analysis but not for actual execution because in Java all objects must have their constructor called to initialize memory.

The invokes that are not included in the slice need to be recovered. This recovery phase finds the corresponding invoke to init for all included NEW instructions

and includes them if they are not in the slice already.

3.4.2.6 Recovery Phases and Fixed Point

The afore mentioned recovery phases can lead to the inclusion of new elements that need the operation of other phases to be applied again.

For example, a super constructor's inclusion may require another object's constructor which may in turn require it's super constructor and potentially in turn require another constructor and so on and so forth. Or, consider that any one of these phases may have dependencies that cause the inclusion of new instructions that have exception handlers.

To handle this, the phases all run and repeat until a fix point is reached; that is, a full run of all phases where no changes are no longer made. To handle this, changes made to the final slice are recorded in a transaction log.

3.4.3 Generate Executable

The WALA shrike instrumentor is used to modify the original jar to reflect the changes described by the final slice operation map produced by the complete slice generation.

The final slice map is a maping from each intsruction to a bytecode-manipulation operation. All instructions are iterated through and the corresponding action in the final slice map is applied. The operations act as follows:

1. For every instruction that is marked as *included*, the instruction is left alone to appear in the final jar as is.

2. For any instruction that is marked as *omitted*, the shrike instrumentor is used to replace said instruction with an empty code block, thus removing

the instruction from the final jar.

3. For any instruction that is marked as *replacewithpop*, the shrike instrumentor is used to replace said instruction with a the PopInstruction.

Finally, the shrike passes are terminated and the output jar is produced.

Chapter 4

Experimental Results and Analysis

In this chapter, we present some results and observation from experiments to study the quality of the WALA slicer and correctness of our executable slicer implementation.

4.1 Benchmarks

Certain limitations in the current implementation of the original WALA slicer prevented the use of larger/standard benchmarks to evaluate our executable slicer. We explain these limitations in Section 5.1. Therefore, we constructed our own benchmark set consisting of smaller but realistic programs to test our project. Our slice criteria allowed the majority of the program to appear in the slice.

We include the following programs in our benchmark set. (a) QuickSort: implementation of the quicksort sorting algorithm, (b) HeapSort; implementation of the heapsort sorting algorithm; (c) MergeSort; implementation of the MergeSort

sorting algorithm, (d) TransposeAMatrix; program to compute the transpose of an aribtrary matrix, and (e) TSPNearestNeighbor; program to compute the solution of the traveling salesman problem using an implementation of the nearest neighbor algorithm. These programs implement standard algorithms in Computer Science.

4.2 Slice Comparisons

	Full Trace		App Only	
Benchmark	Original	Static Sliced	Original	Static Sliced
QuickSort	599,614	574,137	894	840
HeapSort	64,328	33,802	833	727
MergeSort	59,616	39,988	1,585	1,536
TransposeAMatrix	1,732,431	1,593,387	4,574	4,235
TSPNearestNeighbor	1,335,005	1,324,365	5,832	5,811

Table 4.1. Trace size comparisons.

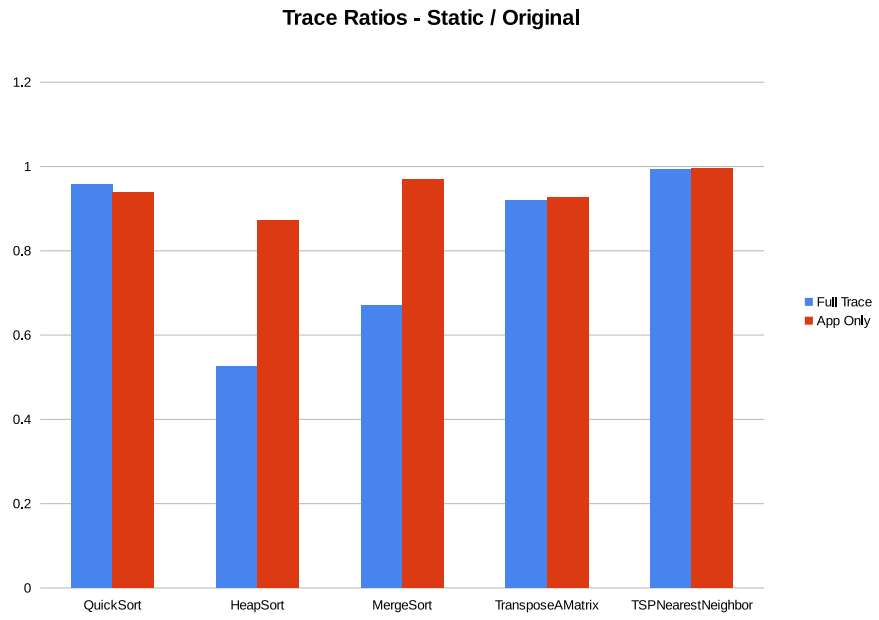


Figure 4.1. Trace Ratios - Statically Sliced Trace Size / Original Trace Size.

The Hotspot VM was used to generate traces of the original and statically sliced benchmarks for comparison. The traces instruction counts are compared above in Table 4.1. The trace size ratios are shown in Figure 4.1; the bars show the ratio of the size of the statically sliced program trace to the size of the original program trace.

Most of the statically sliced program traces were 90% or greater the size of the original program trace for both the full trace and the application only subset. This is as expected as the benchmarks are simple algorithms with little sideffects; id est, most of the program should be needed in the static slice to produce the correct output.

The HeapSort and the Mergesort deviated from this expectation with the full trace of the sliced program at 52.55% and 67.08% the size of the original program traces, respectively. The application only traces however are back in the 80-100% range. With these two benchmarks there are a few program statments that produce inconsequential output; id est, output not needed in the static slice. These statements are invokes inside the application classes to functions outside the application classes (mainly System.out.println()). The invokes themselves comprise a very small portion of the application classes trace but their callee functions comprise a very large portion of the full trace; which is why we see most of the instructions appearing in the application only trace but not the full trace.

4.3 Performance of Executable Slice Extensions to WALA

	Heap Usage (B)		Duration (ms)	
Benchmark	WALA	All	WALA	All
QuickSort	719,022,344	733,059,118	3,497	3,724
HeapSort	728,685,230	745,328,660	3,191	3,503
MergeSort	744,654,237	757,513,290	3,946	4,084
TransposeAMatrix	712,579,912	717,016,988	3,408	3,467
TSPNearestNeighbor	725,464,171	781,390,128	3,336	3,644

Table 4.2. Maximum utilization of heap and slice computation duration.

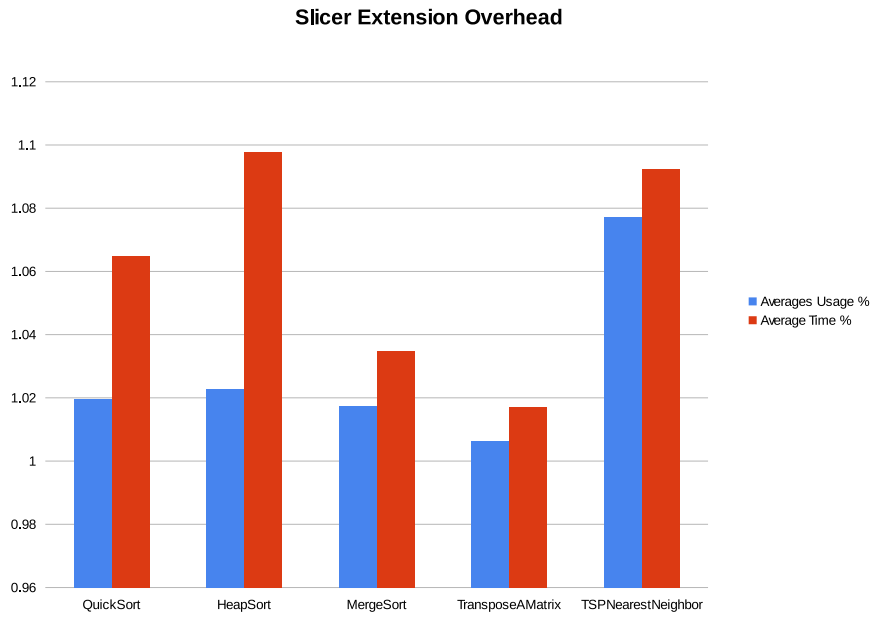


Figure 4.2. Slicer Extension Overhead.

Table 4.2 shows the maxmimum heap usage and the computation duration for compution the WALA structures only and for the WALA structures extended with the executable slice components. The WALA only represents the costs of computing the System Dependence Graph for the input program and the WALA SSA Slice. The All columns represent the WALA components and the additional computation for the non-SSA source table and the generation of the executable

slice.

Figure 4.2 shows the maximum heap usage and the computation duration as a ratio of the extended computation cost to the WALA only computation cost for each of the benchmarks.

The maximum heap usage ranged from 100.62% (TransposeAMatrix) and 107.71% (TSPNearestNeighbor). The computation time ranged from 101.73% (TransposeAMatrix) and 109.23% (TSPNearestNeighbor). The overhead for both metrics for the executable slice extensions were always below 10%. The average overhead accross all benchmarks for heap utilization is 2.86% and for computation time is 6.14%.

Chapter 5

Future Work

One of the eventual goals for the executable slicer, is automatic program partitioning and subsequent program parallelization. The ability to automatically compute executable slices allows a program to be partitioned out by sections based on control and data dependence. Partitioning of this nature can be very useful in optimizing single processor ready programs to make use of multi-processor environments, today's commonplace.

Many programs are still built to be only single processor conscious. The ability to parallelize these programs automatically is highly appealing.

It is often the case that components of single threaded programs delay later executing components that have no data or control dependence relationships. In these cases, there are independent states that can be computed in parallel. Though there will likely always be some overlap among these partitions, the redundant computation of common state will in most cases be offset by the benefits of parallelizing the computation of unique state. Furthermore, the redundant computation of common state can often be limited via various synchronization mechanisms.

5.1 WALA Limitations

WALA SDG builder has limitation when it comes to static variables, aka globals. The SDG builder does not attempt to determine which writes to a specific static variable can affect which reads. The SDG simple includes all read edges to the same node and all writes go out from that node. As such, the SDG seems to suggest that all writes affect all reads, which is not necessarily true. This simplification is represented in the slice as follows: when a read is in the slice, all writes in the entire program to that static variable is included. This reality is very unsatisfying and greately limits the ability for the slicer to cut down the size of any program that uses static variables at all.

The WALA SDG builder suffers a similar limitation when it comes to instances of classes and invokes of their methods. If there are multiple instances of a single class and only truly required, invokes on both may show up in the slice. This happens when a private variable of an instance class is needed for one instance but not all and a method is invoked to alter that private variable; the result is that all invokes to that method are included.

E.g. Class MyClass is instance twice, Instance1 and Instance2. They both have a private instance variable Count. They both have a public method Increment() which increases the value of Count for the given instance. If a single invoke of Increment is included for a single method because of the value Count being important, all calls to Increment on all other objects will be included as well.

This instance class limitation is also quite debilitating as virtually any program that uses instance classes will include many instruciton that do not belong in the slice, thus limiting the usefulness of the slicer.

Chapter 6

Conclusions

We have produced extensions to the WALA slicer framework to compute fully executable slices. A fully executable slice can always be formed within the limitations of the WALA slicer itself. Fully executable slices have been produced for a number of programs and criteria that meet WALA's limitations and correct execution and output of the resulting statically sliced programs have been observed. A few programs were selected, the benchmarks, for analysis. Across these benchmars, the overhead introduced by the WALA slicer for heap utilization averaged at 2.86% and the computation time overhead averaged at 6.14%. This overhead is made as a trade-off for the tracking of all non-SSA information, which makes for a significant portion of the program's data dependence, and the computation of the executable slice using said data dependence.

Limitations of WALA have been documented along the way and are described more fully in Section 5.1.

References

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.*, 23(6):589–616, 1993.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. Technical Report SERC-TR-56-P, Purdue University, 1989.

[3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, New York, NY, USA, 1993. ACM.

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

[5] D. Binkley and K. Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

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

[7] D. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 44, Washington, DC, USA, 2003. IEEE Computer Society.

[8] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[9] M. Daoudi, L. Ouarbya, J. Howroyd, S. Danicic, M. Harman, C. Fox, and M. P. Ward. Consus: A scalable approach to conditioned slicing. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 109, Washington, DC, USA, 2002. IEEE Computer Society.

[10] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 121–134, New York, NY, USA, 1996. ACM.

[11] M. D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 1994.

[12] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–392, New York, NY, USA, 1995. ACM.

[13] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322, 1992.

[14] K. B. Gallagher. *Using program slicing in software maintenance*. PhD thesis, University of Maryland at Baltimore County, Catonsville, MD, USA, 1990.

[15] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance*, pages 299–308, Orlando, FL, USA, 1992.

[16] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 1990. ACM.

[17] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[18] G. Inc. 2002. The codesurfer slicing system. last retrieved from http://www.grammatech.com/products/codesurfer, June 2009.

[19] J. Larus and S. Chandra. Using tracing and dynamic slicing to tune compilers. Technical Report CS-TR-93-1174, University of Wisconsin-Madison, August 1993.

[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] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[22] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[23] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[24] M. Weiser. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters*, 17(3):129–135, 1983.

[25] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1979.

[26] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[27] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 94–106, New York, NY, USA, 2004. ACM.

[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.