Robert Oehlmann

Static Single-Assignment for Program Slicing
on Binary Intermediate Language

May 21, 2013

supervised by:
Prof. Dr. Sibylle Schupp
Dipl.-Ing. Arne Wichmann

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.


Robert Oehlmann
1 Introduction

Understanding software from machine code is a difficult task and still hardly supported by tools. The available tools used in the industry are capable of disassembling binary files and detecting large portions of the code in it. They do, however, provide little support on a semantic level.

Working from machine code is still necessary, for example when maintaining aging embedded system. If the original source is no longer available, understanding the machine code, extracted from a functional system, is often the only way to develop maintenance procedures.

The idea in this bachelor thesis is to reproduce one common step of program understanding, usually done by a human software engineer, in a software tool, to thus reduce the amount of manual work. We will be using Program Slicing to accomplish this task. Program Slicing is the reduction of a program to only those statements relevant for a specific behavior.

Research has shown that software engineers, when working with unfamiliar code, spend large amounts of their time tracing variables through the program and determining which parts of the program influence a certain value. By utilizing Program Slicing, a program can automatically be reduced to only those statements affecting a specific variable.

Machine code — as the name suggests — is specific to each hardware platform. Disassembling a binary file converts the unstructured binary input into assembler code, clearly separating operators and operands. However the semantics of the assembler statements are different on each hardware platform and the effects are rarely obvious from the mnemonics.

For a software engineer unfamiliar with the hardware platform at hand, this implies keeping the manual open while reading the assembler code. For a software tool to perform analysis on machine code, this requires us to give the tool access to the semantics. We could do so by hard-coding the semantics into the tool, making it specific to a certain hardware platform. Due to the large amount of platforms in use, this would however diminish the usefulness of the tool immensely.

To cope with a large variety of hardware platforms, we do not directly work on machine code, but on Binary Intermediate Language instead. Binary Intermediate Language (BIL) expands assembler code by stating the semantics of its statements explicitly. By using the BIL representation of a program as input instead of the machine code, the tool can be platform agnostic.
In order to apply Program Slicing for tracing values through a program we also need some kind of def-use information. Static Single-Assignment form makes def-use-chains obvious by renaming variable such that every variable is just assigned once. This is usually done by adding an index to the variable names. At merge points in the control flow graph, special $\phi$-functions are introduced. They use all possible previous instances of the variable and creating a new definition.

This bachelor thesis explores the application of Program Slicing to code in Static Single-Assignment form. Furthermore, it discusses slicing on multiple language levels, as the algorithm operates on platform independent Binary Intermediate Language code, but the result is displayed in assembler code. Goal of this bachelor thesis is the implementation of a tool that performs this kind of Program Slicing. It provides algorithms for transformation to Static Single-Assignment form, for forward and backward Program Slicing, as well as a graphical user interface that makes the results easily accessible.
2 Background

This chapter gives an introduction into the theoretical background of the techniques used in the following implementation.

To clearly differentiate between the different language layers we refer to assembly-level statements as “instructions” and use the term “statements” exclusively for statements in BIL.

2.1 Control-Flow Graph

A control-flow graph (CFG) is a directed graph, used to represent all possible execution paths of a program. One or more program statements make up the nodes of the graph, the edges indicated possible orders of execution.

1: read(X)
2: if \( X < 10 \) then
3: \( X = X \times 2 \)
4: end if
5: write(X)

Figure 2.1: Code example (left) and the corresponding control-flow graph (right)

The execution of the code example shown left in Figure 2.1 always starts with a read instruction and a subsequent if condition (l. 1–2). If the condition is satisfied, the execution continues with the body of the if-clause (l. 3), otherwise the body is skipped and the next statement after the if-clause is executed. In the CFG (right) this is indicated by two outbound edges of the if-condition. The execution will then always finish with the write instruction (l. 5). The two possible predecessors are indicated by two inbound edges in the CFG.

Reducibility Control-flow graphs are reducible, if they can be reduced two a single node by the application of two reduction rules. The first rule states to replace self
loops by a single node. Replacing two subsequent nodes, with the first being the only predecessor of the second, by a single node, is the second rule. The rules are visualized in Figure 2.2.

![Diagram showing two rules applied to test reducibility of a graph](image)

Figure 2.2: The two rules applied to test reducibility of a graph

2.2 Binary Intermediate Language

Intermediate languages are not a new concept, in fact they are commonly used in compilers. The goal of an intermediate language representation is to state all (side-)effects explicitly and platform independently, in order to simplify further optimizations or analyses.

The Binary Intermediate Language for this work is based on an intermediate language developed by David Brumley et al., for their project Binary Analysis Platform (BAP). It is specifically designed for software analysis, which makes it a good choice for this work. A formal definition of the language is given in the BAP manual [1].

```plaintext
addr 0x0 @asm "add %eax,%ebx"
label pc_0x0
  t:u32 = R_EBX:u32
  R_EBX:u32 = R_EBX:u32 + R_EAX:u32
  R_CF:bool = R_EBX:u32 < t:u32
  R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EBX:u32 ^ t:u32 ^ R_EAX:u32))
  R_OF:bool = high:bool((t:u32 ^ ~R_EAX:u32) & (t:u32 ^ R_EBX:u32))
  R_PF:bool = ~low:bool(R_EBX:u32 >> 7:u32) - R_EBX:u32 >> 6:u32 -
               R_EBX:u32 >> 5:u32 - R_EBX:u32 >> 4:u32 - R_EBX:u32 >> 3:u32 -
  R_SF:bool = high:bool(R_EBX:u32)
  R_ZF:bool = 0:u32 == R_EBX:u32
```

Figure 2.3: Example of Binary Intermediate Language

The BIL code in Figure 2.3 represents the x86 assembler instruction `add %eax, %ebx`, the addition of two extended registers. The first statement `addr 0x0` denotes the start of a new assembler instruction and preserves the original address of the instruction. The rest of the statement beginning with `@` is an annotation, used here to
also preserve the original assembler instruction. Similar to an assembler label, the label statement in BIL marks a potential jump target. The following assignment statements contain the semantics of the assembler instruction.

This examples shows how the `add` instruction, which only references two registers, influences a lot of flags as well. Variables are used to represent registers, they are however not limited by the number of registers, thus temporary variables can be introduced as needed.

### 2.2.1 Representing Memory Access

The BIL version documented in the BAP manual provides two functions to represent memory access: `load` and `store`. In general, memory is represented as a single variable. The `load` and `store` expressions are used to address certain parts of the memory.

Loading a value from memory is done by applying `load` to a memory variable, an address, the endianness and access width. The function returns the value at the given address in the memory variable, in accordance with the endianness of the platform, thus providing a memory abstraction.

`store` takes the same parameters as `load` and additionally the value to store. The return value is a new memory state.

### 2.3 Static Single-Assignment Form

Using Static Single-Assignment form (SSA), the relation of variable uses (reading a variable) and variable definitions (setting a variable) can be explicitly stated by renaming variables such that each variable is just defined once. Like intermediate languages, SSA is commonly used in compilers, in order to simplify optimizations and determine register allocations.

New variable names are commonly constructed by adding a numeric index to the existing variable names. Each definition of the same variable will yield a new instance and thus a new index. The following uses are updated accordingly.

#### φ-functions

At merge points of the control-flow graph (CFG), possibly multiple previous definitions exist. In that case φ-functions are introduced. A φ-function uses all active definitions of a variable and defines a new instance of that variable. There is no algorithmic semantic to φ-functions, they are just used for analytical purposes, in order to represent multiple previous definitions of a variable.

An example for the transformation to SSA is given in Figure 2.4. The original program is given on the left, two different SSA versions are shown in the middle and on the right. In both SSA results, all variable references from the original program are
Figure 2.4: Example program (left) transformed to Static Single-Assignment form with non-necessary $\phi$-functions (middle) and in minimal form (right); based on an example from Aycock and Horspool (2000) [2]

renamed by adding an index and $\phi$-functions are introduced. While the SSA code in the middle has $\phi$-functions for every variable at the beginning of each CFG block, they are used sparingly in the code on the right. Actually the right hand side code contains only those $\phi$-functions necessary to produce a correct solution, hence it is in minimal form.

2.3.1 Minimal Form

When SSA is used for optimization, inserting $\phi$-functions where they are not necessary, might lead to missed optimizations. Thus the notion of the minimal form has been introduced. The most common definition states that the SSA representation is in minimal form, when the number of $\phi$-functions is minimal. $\phi$-functions are not necessary if they have just one argument or all arguments can be shown to be equal.

Figure 2.4 shows an example program (left) next to its non-minimal (middle) and minimal (right) SSA form. The statements of the form $i_x \leftarrow \phi(i_y)$ in the non-minimal
form (lines 10, 11, 14, 15, 18, 19) can easily be identified as non-essential, since they just perform variable renaming. After removing those and reversing the renaming of the associated instances, another non-essential $\phi$-function can be identified in line 7 of the non-minimal form: $j_2 = \phi(j_1, j_2)$. That statement originally reads $j_2 = \phi(j_1, j_5)$, but the reversed renaming reveals that the second arguments is inherently the same variable instance that is defined in the statement: $j_5$ is defined from $j_3$ in line 19, $j_3$ is defined from $j_2$ in line 11. Hence the statement is again just instance renaming and thus non-essential.

For our Program Slicing approach, minimal form does not improve precision. Extraneous $\phi$-functions will require more definition-lookups in our algorithm (see section 2.4), but will not include additional (ghost) instructions into the output slice. Thus minimal form is not a requirement for Program Slicing, it can however improve the performance of the Program Slicing algorithm.

### 2.3.2 Constructing SSA

Different algorithms for the transformation of code into SSA form exist, three approaches will be discussed in the following sections.

#### Dominance Frontier

A variety of algorithms employs dominance information in order to determine the necessary $\phi$-functions [4, 5].

In a directed graph, a node A dominates a node B, if all paths from the start node to node B run through node A. If A and B are different nodes, A strictly dominates B.

The dominance frontier for A is defined as all nodes C that have a predecessor B, which is dominated by A, but are not strictly dominated by A themselves. When visualizing the region of the control-flow graph dominated by a node (compare Figure 2.5), the region has a border. We call all those nodes just outside that dominated area as belonging to the dominance frontier.

Figure 2.5 depicts an exemplary CFG. Nodes 4 and 6 are dominated by node 3. The part of the graph dominated by node 3 is highlighted gray. Nodes 5 and 7 form the dominance frontier of node 3, as they have a predecessor that is dominated by node 3 but are not dominated by node 3 themselves. This becomes obvious in the figure as their inbound relations $4 \rightarrow 5$ and $6 \rightarrow 7$ cross the border of the (imaginary) dominated area.

The dominance frontier is important for SSA construction, because the nodes of the dominance frontier of A (any node other than the start node) can be reached from a variable definition in A, as well as previous definitions (a previous definitions always exists, because all variables are initiated in the start node). Thus $\phi$-functions for this variable are required at the beginning of the nodes of the dominance frontier.
One algorithm that facilitates the dominance information for the placement of $\phi$-functions is given by R. Cytron et al. (1991) [4]. For each variable it starts with a work list of all control-flow graph nodes that contain a definition of that variable. Definitions are taken from the work list and $\phi$-functions are introduced at its dominance frontier, if they do not yet exist. New $\phi$-functions yield new definitions that are added to the work list. The algorithm proceeds with the next variable, when the work list is empty.

The iterative insertion of $\phi$-functions will always terminate, as each iteration step removes one item of the work list. New items are only added if a $\phi$-function is introduced. Since only one $\phi$-functions is added per graph node (and variable) and the graph is finite, the iteration terminates.

A more thorough explanation of the computation of dominance frontiers and the subsequent transformation to SSA form is given by R. Cytron et al. (1991) [4].

**Really Crude Approach**

The most simple approach to create SSA for a non-linear control-flow, called “really crude approach” (RC-Approach) by Appel [3], is to introduce $\phi$-functions for all variables at the beginning of every CFG block. The $\phi$-functions are constructed such that the k-th argument is the last instance of a variable at the end of the k-th control-flow predecessor.

The generated SSA is very verbose and yields a large number of variable instances that are actually equal (compare Figure 2.4).

**Minimizing RC-Approach**

Based on the RC-Approach by Appel [3], the idea of the algorithm proposed by J. Aycock and N. Horspool [2] is to remove the extraneous $\phi$-functions, transforming the “really crude” SSA into its minimal form for reducible CFGs.
The algorithm consists of two phases, the first being the application of RC-Approach mentioned above, the second being the minimization phase. During the minimization phase, \( \phi \)-functions of the form
\[
V_i \leftarrow \phi(V_i, V_i, \ldots, V_i)
\]
are removed and statements of the form
\[
V_i \leftarrow \phi(V_{x_1}, V_{x_2}, \ldots, V_{x_n}), \text{ with } x_1, x_2, \ldots, x_n \in \{i, j\}
\]
are removed, with all mentions of \( V_i \) being renamed to \( V_j \). Those operations are carried out iteratively, until a fixpoint is reached.

The correctness of the algorithm has been proven by Aycock and Horspool. They also prove that this algorithm yields SSA code in minimal form for all programs with reducible control-flow graph.

Our SSA implementation is based on this algorithm. The algorithm excels at simplicity, since it does not require the computation of the dominance tree and dominance frontiers. While it does not create minimal form SSA code in all cases, the authors argue that most CFGs are reducible and minimal form is not required for our desired use case of Program Slicing.

### 2.4 Program Slicing

Program Slicing is a technique to automatically extract a subset of a program that is related to a specific behavior. We call the resulting subset slice. One common way of Program Slicing is to extract all statements that affect the value of a variable at a certain position in the code. This is called static backwards slicing.

1: \( x \leftarrow 2 \)  
2: \( y \leftarrow 5 \)  
3: \( z \leftarrow 4 \)  
4: \( k \leftarrow y + z \)  
5: \( z \leftarrow x + y \)

Figure 2.6: Example of Program Slicing; original program (left) and program slice for variable \( z \) in line 5 (right)

In the above example (Figure 2.6) the original program has been sliced for the variable \( z \) in line 5. Since the previous definition in line 3 and use of \( z \) in line 4 do not influence the value of \( z \) in line 5, they do not belong to the resulting program slice. In line 5, \( z \) is defined from \( x \) and \( y \) and thus the most recent definitions of those variables (in line 1 and 2) belong to the program slice.
**Slicing Criterion**  The slicing criterion specifies the behavior to be extracted into the slice. This is usually a tuple \( \langle i, v \rangle \) with \( i \) being the point of interest and \( v \) being a variable or a set of variables to analyze for. In the above example (Figure 2.6) the slicing criterion is \( \langle 5, \{z\} \rangle \), expressing that everything that affects the value of the variable \( z \) at line 5 should be part of the slice.

**Slicing Concepts**

Program slicing has been proposed as early as 1979 by Mark Weiser. His first method, which he elaborated more formally in 1981 [6] and which we are now referring to as static backward slicing, has since been accompanied by a number of other slicing concepts. More specifically, backward slicing has a counterpart in forward slicing now and dynamic slicing is an extension of the original static slicing. M. Harman and R. Hierons have given a good overview of the Program Slicing landscape in 2001 [7], the following section only gives a short conceptual introduction in those four most important developments.

**Backward slicing**  creates slices that contain all those statements which affect the value of a variable at a specific position, as discussed above. The slices is constructed by tracing variables back against the direction of the control-flow, hence the name.

**Forward slicing**  in contrast follows the direction of the control-flow and creates slices that contain all statements affected by a variable at a specific position.

**Static slicing**  is an offline analysis, performed only on the code.

**Dynamic slicing**  additionally exploits information gained during the execution of the code. For example when an unexpected value occurs, a slice could be constructed for the specific input that lead to the bug.

The implementation of this work creates forward and backward slices using static slicing.
3 Implementation

Goal of this bachelor thesis is the implementation of a tool that performs Program Slicing on binary intermediate language, transformed to Static Single-Assignment form. This chapter explains how the presented methods work together and how they are implemented.

The tool is aimed to be user friendly, more specifically it should offer a graphical user interface (GUI) and hide unnecessary complexity. Microsoft Windows is a common operating system among the potential user group, hence our tool should run on this platform.

From those requirements C#, with Windows Presentation Foundation (WPF) for UI design, was chosen as the development platform.

3.1 Workflow

When loading a binary intermediate language (BIL) file, the tool starts processing the input by parsing the BIL into an abstract syntax tree (AST). From the AST the assembler instructions and the control-flow graph (CFG) are extracted. The code is then transformed to Static Single-Assignment form (SSA). Information about variable
instances, as well as definitions and uses of those is kept for later use. The CFG is then displayed in the main area of the tools user interface as a graph. The CFG nodes display the code of the program under analysis on an assembler instruction level. The underlying BIL code is not part of the graph display.

As soon as the CFG is displayed, the user can interact with the program. In the user interface, shown in Figure 3.1, the graph is displayed in the main area (1). When selecting a CFG node area 2 is used to list all contained assembler instructions, while the list of φ-statements at the beginning of this block is displayed in area 4 and the variable instances yielded are listed in area 3. By selecting an assembler instruction in area 2, the user can see the BIL representation of the instruction in area 4 and a list of all BIL variable instances used and defined within this instruction in area 3. The user can pick one of the variable instances in area 3 as slicing criterion and trigger the slicing. The computed slice will be displayed in the main area.

As this chapter explains the individual steps of the workflow, they are applied to an ongoing example. The example code is part of a task dispatcher of program running on an embedded system.

### 3.2 Data Structures

![Figure 3.2: Classes for CFG and SSA representation, and their relations](image)

The above Figure 3.2 gives an overview of the C# classes used to store the CFG and the references between the classes. Further temporary data structures are used during single steps of the workflow, however this is the main data structure kept throughout the usage of the tool.

Different parts of the main data structure are relevant during the different phases of the workflow.
**CFG Construction**  The data structure is initialized by constructing a new *CFG* object, which acts as a container element to the subsequent classes and as such contains the representation of a complete program. Furthermore the *CFG* class contains the main algorithms and interfaces to the GUI. When a new *CFG* object is constructed from the abstract syntax tree (AST) of a loaded BIL file, all of the classes, except *VarInstance*, are used to represent the loaded program.

**SSA Transformation**  During the transformation to SSA form, objects of the class *VarInstance* as well as the class *Statement* are constructed to represent the multiple instances of variables and the newly introduced φ-function statements. All other classes are referenced.

**Program Slicing**  Program slicing is only operating on a rather small subset of the main data structure: Starting from a variable instance, it looks up the statement of its definition and then goes back from that statements variable uses to determine other variable instances that need to be traced.

Further explanations of the classes will be given in the following sections, as they become relevant.

### 3.3 BIL Parser

The BIL parser is created by the ANTLR parser generator, a basic grammar file has been available to me thanks to previous work of my advisor. Modifications to the grammar and assumptions about the input BIL files are explained in this section.

**Annotations**  The tool expects additional information that is not usually contained in BIL files. These can be added in the form of annotations, which are attached to statements. Annotations have the form `@label "content string"`, with the *label* as identifier for an annotation and its content in form of a string. Any arbitrary identifier can be used as label.

#### 3.3.1 Assembler Instructions

Transforming machine code into BIL makes the code much longer and less readable, hence it is desirable to display the program under analysis on an assembler level instead of BIL level. Users who have previously been working with disassemblers will be familiar with this level of abstraction. And while BIL code can be helpful in understanding the semantics of a single instruction, it makes it hard to see higher level behaviors.
In order to be able to display the program in assembly code, the assembly instructions need to be contained in the input file. The BIL statement `addr` marks the beginning of each new assembler instructions, thus the tool expects an annotation with the label `asm` here to contain the assembler instruction.

### 3.3.2 φ-functions

When performing SSA, φ-functions need to be introduced in the language (see section 2.3). Since the BIL definition does not contain a language element suitable to represent them, we chose to extend the language by the keyword `Phi` which is used as a function with a possibly empty set of arguments. The arguments are noted in parenthesis and comma separated, resulting in statements like `x__3 = Phi(x__1, x__2)` and `x__0 = Phi()`.

The empty `Phi` function is used to represent the initialization of a variable in a CFG node with no predecessors, commonly denoted as assigning ⊥ to the variable in scientific papers. The initialization of variables is important to assure the correct insertion of φ-functions in the rest of the code. Since we perform an intra-procedural analysis, a previous definition of a variable may always exists and the analysis should respect that by initializing all variables.

Missing initializations might result in missing φ-functions. For example if a procedure contains just one definition of register X, but the control-flow permits to bypass this definition, the minimization algorithm will remove all φ-functions. A later use of register X would only be linked to that single definition of X, although a different definition outside of that procedure could be relevant. By initializing register X with ⊥, φ-functions at the dominance frontier of the definition of X indicate the two possible definitions and the ⊥ symbol represents a definition outside the procedure.

### 3.3.3 Jump and call targets

Except for explicit manipulation of the program counter, BIL provides two statements to represent non-linear control-flow, one being the unconditional jump `jmp`, the other being the conditional jump `cjmp`. In order to express subroutine calls, BIL statements just before the jump could be used to push the return address on the stack, hence store the return address in a variable that represents the memory and modify the stack pointer variable. A return instruction could then modify the stack pointer again, read the return address from the memory variable into another variable and use it as the target argument for the return jump.

While this is a correct representation and could potentially allow the execution of BIL, it makes it hard for a static analysis to differentiate between plain jumps and subroutine calls, because the jump statement may be the same for them and only the semantics of the preceding statements make the difference.
Furthermore the jump or call target may not be unique, for example, if it’s an indirect jump, based on a register value.

Those information can be retrieved with analyses that are outside the scope of this bachelor thesis, but available in other tools. This tool will expect the information as annotations to all jump statements. The label of this annotations is either jmptargets or calltargets, determining whether it’s a jump or a call instruction, and the content string contains a comma separated list of possible jump targets as addresses in hexadecimal format.

3.4 CFG Construction

Once the BIL file is read in, the AST will be passed to the constructor of the CFG class which will then aggregate the BIL statements into the original assembler instructions and construct the CFG.

A simplified version of the algorithm is given as Algorithm 1 below. Line references in the following paragraphs refer to this algorithm.

3.4.1 Assembler instructions

Grouping the BIL statements into the original assembler instructions is a rather simple operation, done by iterating over all BIL statements, collecting them into an instruction object and starting a new instruction whenever an addr statement is reached, which marks the beginning of a new instruction in BIL.

More precisely, the tool iterates over all statement ASTs returned by the BIL parser. To allow further information to be stored with the statement (such as variable references), the statement AST is first wrapped in an object of the class Statement (l. 5), before being appended to the current Instruction object (l. 42). If the statement is of type addr (l. 6), a new instances of Instruction will be constructed before appending the statement (l. 20).

Figure 3.3 shows three instructions from the task dispatcher example and their respective BIL representation. The tool starts from the right hand side BIL code and groups them into assembler instructions (left) as part of the CFG construction.

3.4.2 CFG blocks

In order to keep the computational complexity low, the CFG is constructed alongside the aggregation of assembler instructions in one loop. Similar to the Instruction objects, which collect the BIL statements, instances of the class CFGBlock are used to collect the instructions. Additionally, two dictionaries are used as temporary data structures: One matches code addresses to CFG blocks (not part of the simplified algo)
Figure 3.3: Part of the task dispatcher example code on instruction level (left) and as BIL code (right)

Read Assembler Instructions  We construct the CFG on assembler instruction level, while iterating over BIL statements, so most of the CFG construction algorithm is only executed when an `addr` statement signals the beginning of new assembler instruction (ll. 6–32). However information on how to proceed at an instruction break is collected from intermediate BIL statements, most notably the jump statements `jmp` and `cjmp` (ll. 33–37). Upon reaching either statements, the jump type and list of jump targets of the containing `Instruction` object is updated from the attached annotation (compare ll. 34, 35 and section 3.3.3).

Determine Block Breaks  At an instruction break, two criteria indicate the requirement to also start a new CFG block: Either the previous instruction is a jump or call instruction (ll. 9–19) or the new instruction is a jump target listed in the jump dictionary mentioned before (ll. 23–31). In case a new block is started, the previous instruction is stored in the previous block (l. 8), which is in turn stored to the containing CFG object (ll. 10, 25).

Insert Block Links  The blocks are then linked depending on the origin of the block break. If it was only introduced due to inbound jumps for the next instruction, a fall-
through link between the previous and the new block (l. 28) as well as the the listed inbound jumps are inserted (l. 30). If the break was introduced due to an outgoing jump at the end of the previous block, links are only introduced according to the jump targets (l. 12).

**Lookup and Store Jump Targets** The block dictionary is used to lookup the target blocks from the jump addresses. If a jump address is not contained in the dictionary, it’s most likely a reference to code ahead of the current position and the jump will be inserted into the jump dictionary for later use. If the jump address is in the midst of an already existing block that block will be split and a fall-through link will be inserted to preserve the previous control-flow (not part of the simplified algorithm).

### 3.4.3 Variables

As a preparation for the following SSA transformation, the tool traverses the AST of all assignments (ll. 39–40) and jump statements (l. 36) for variable references. A new instance of `VarReference` is created for each variable reference found.

The first time a variable is met, an object of class `Variable` is constructed and appended to the CFGs dictionary of variables. All `VarReference` objects contain a reference to the matching `Variable` object, the AST subtree they emerged from and the `Statement` object containing them. The statement, in turn, contains a list of all its variable references (compare section 3.2).

With that, the CFGs data structure is complete, employing all classes mentioned before, but `VarInstance`, which will only be used for SSA. The result could already be displayed in the user interface (in the implementation it is only displayed after the SSA transformation has been performed). The CFG of the dispatcher example is depicted in Figure 3.4.
Figure 3.4: Control-flow graph of the task dispatcher example, as displayed in the user interface
Algorithm 1 Construction of CFG from BIL parser output

Require: stmts ← list of statement ASTs

Require: cfg ← new CFG()

1: pendingJumps ← ∅
2: inst ← new Instruction()
3: block ← new CFGBlock()
4: for all AST in stmts do
5: stmt ← new Statement(AST)
6: if stmt.AST.type = ADDR then
7: if inst.statements.Count() > 0 then
8: block.instructions.Add(inst)
9: if inst.jumpType! = NoJump then
10: cfg.blocks.Add(block)
11: if inst.jumpType = Jump then
12: Link block to all inst.jumpTargets
13: end if
14: lastBlock ← block
15: block ← new CFGBlock()
16: if inst.jumpType = Call then
17: linkBlocks(lastBlock, block)
18: end if
19: end if
20: inst ← new Instruction()
21: end if
22: inst.Address ← parseHex(stmt.AST.children[0])
23: if ∃⟨fromBlock,toAddr⟩ ∈ pendingJumps, toAddr = inst.Address then
24: if block.instructions.Count() > 0 then
25: cfg.blocks.Add(block)
26: lastBlock ← block
27: block ← new CFGBlock()
28: linkBlocks(lastBlock, block)
29: end if
30: Link all inbound jumps to block
31: end if
32: end if
33: if stmt.AST.type ∈ {JMP, CJMP} then
34: inst.jumpType ← Jump or Call, depending on annotation
35: inst.jumpTargets ← List of target addresses from annotation
36: Traverse stmt.AST for variable uses
37: end if
38: if stmt.AST.type = ASSIGN then
39: Save variable definition in stmt.AST.children[0]
40: Traverse stmt.AST.children[1] for variable uses
41: end if
42: inst.statements.Add(stmt)
43: end for
44: block.instructions.Add(inst)
45: cfg.blocks.Add(block)
3.5 SSA Transformation

The SSA implementation is based on an algorithm by J. Aycock and N. Horspool [2], explained in section 2.3. It consists of three phases, two of which have been mentioned before.

3.5.1 RC-Phase

The first phase stems from the “really crude approach” and inserts $\phi$-functions for all variables at the beginning of each CFG block. However it employs the improvement suggested by the authors to complete the phase in a single pass.

By making sure that the index of the last variable instance in a block matches the block number, $\phi$-functions can be inserted, even if not all predecessors of a block have yet been processed. A $\phi$-function is assumed to use all previous variable instances, indicated by the index, as arguments. With linear indexing this would not immediately be possible, as the last instance index of a block would only be known after that block has been transformed.

In this implementation the instance index consist of the block number, an underscore as separator and an in-block index starting at 0. The last index inside a block is than renamed to just the block number without the trailing underscore and index.

Due to the data structure, this phase consists of a layered iteration of all statements, beginning with the CFG blocks as the outermost layer.

Inserting $\phi$-functions

In a first step, the $\phi$-functions are inserted. Since each statement and variable reference contains an AST reference and the generated SSA is supposed to be savable, AST trees are constructed. For each predecessor of a block, the last instance is looked up by the blocks index. If a predecessor has not been transformed to SSA, the variable reference is just stored in the Variable object as pending use. For blocks with no predecessor the $\phi$-function is inherently empty, representing the initialization of a variable with an undefined value ($\bot$).

Since the $\phi$-function statements do not belong to any assembler instruction, they take a special role in the data structures. All other BIL statements are grouped in Instruction objects, which are children to the CFG blocks. One could consider to introduce a dummy instruction that contains the $\phi$-function statements, however instructions are required to have an address. Since free address spaces at the beginning of blocks can not be assumed and would yield other problems (e.g. jump target addresses), these statements will be child directly to the CFGBlock object in its initStatements property.

Variable instances are represented by the VarInstance class, which consists of one VarReference as definition, a list of VarReference for all uses, an index and a reference
to the Variable object. Inserting a $\phi$ function yields a new instance and adds uses to all predecessor instances. New VarInstance objects are also added to the instance dictionary of the superior Variable object with the index as key to allow later lookup.

**In-block SSA**

The second step on CFG block level is the nested iteration over the assembler instructions of the block, the BIL statements of an instruction and the variable references of a statement.

The further proceeding depends on the type of a variable reference. Variable uses are simply linked to the current instance. At variable definitions a new VarInstance is constructed and replaces the current instance of that variable.

**Last Instances**

To comply with the assumption that the last instance of a variable inside the block has the block number as index, all last instances are now renamed. During this third step, we will also check whether any pending uses of the instances with this index have previously been noted. In that case we can now link those uses to the variable instance.

As a result of this phase we have collected all information for a correct, but not necessarily minimal, Static Single-Assignment form. The variable instances have been determined and are linked to their individual definitions and uses and all $\phi$-functions have been inserted. The BIL code however is not yet in SSA form, as no variable reference in the code has yet been renamed. This will only be done in the third and last phase of the SSA transformation because a large number of variable references will be assigned to different instances in the following minimization phase. Performing code level renaming for every change would be a large waste of performance.

**3.5.2 Minimization Phase**

According to the algorithm, minimization is conducted by iteratively going through all $\phi$-functions and removing the unnecessary statement.

**Iterating over the $\phi$-functions**

One could come up with two ways of iterating over all $\phi$-function statements: Iterating over the initStatements of all CFG blocks or iterating over the variable instances of all variables and filtering out those defined by a $\phi$-function. Although the latter might appear less efficient on first sight, since actual variable definitions that are inherently excluded by the first approach have to be exclude manually, it has one advantage which could make it more efficient under certain conditions.
The deletion of $\phi$-statements is not done in a single pass, but is done iteratively until a fix-point is reached. The first approach will have to check all $\phi$-functions of all variables in every iteration until a fix-point is reach for all variables. With the second approach however, it is possible to do the iterations on a per-variable basis, iterating over the $\phi$-statements of one variable until a fix-point has been reached for that single variable. The remaining $\phi$-statements will not be check during the iterations over any other variable.

This is beneficial if large number of variables reach a fix-point after a single iteration, while few variables require a large number of iterations. Furthermore the check, whether a definitions stems from a $\phi$-function, in the second approach, is quite cheap compared to the check, whether a statement can be deleted, that is performed unnecessarily often in the first approach. Thus this implementation is based on the second approach.

**Combining variable instances**

For each variable instance that is spawned by a $\phi$-function, the variable instances used as arguments are analyzed. When the statement matches the form

$$V_i \leftarrow \phi(V_{x_1}, V_{x_2}, ..., V_{x_n}), \text{ with } x_1, x_2, ..., x_n \in \{i, j\}$$

just one distinct instance ($V_j$) is used in the arguments, next to the instance that is defined ($V_i$).

In that case, the used instance $V_j$ absorbs the defined instance $V_i$ by taking over all its variable uses. The statement is then removed from the blocks $initStatements$ and the variable instance $V_i$ is deleted.

Whenever a minimization of this form is conducted, the update counter of the iteration is incremented. The iteration continues until no updates have been performed, hence a fix-point is reached. Since the number of $\phi$-functions is limited and iteration is only continued when a $\phi$-functions is removed, it will always terminate.

The second minimization rule by J. Aycock and N. Horspool, removing all statements of the form

$$V_i \leftarrow \phi(V_i, V_i, ..., V_i)$$

is not applied in this tool. Since the variable instance $V_i$ is spawned by this statement, removing it would leave all uses of $V_i$ without definition. This conflicts with the fundamental assumption that every variable instances is defined, at least with $\bot$. This assumption is important to correctly determine the placement of $\phi$-functions with respect to variable definitions outside the scope of the analysis. Furthermore, due to the fact that all variables get initialized with $\bot$ at CFG blocks with no predecessor, statements of this form could only occur at blocks which only have back edges as inbound edges and no sole predecessor.
In order to sustain the assumption that every variable is defined, statements of given form could only be replaced by \( V_i \leftarrow \bot \). Since this replacement is not very beneficial, it’s not performed by the tool.

### 3.5.3 Renaming Phase

Once the minimization phase is completed, the indices of variable references will not be changed further and thus we can show the results in the BIL code and finally transform it to SSA form, by renaming all variable references with an index. To avoid further modifications of the BIL, the index will simply be appended to the variable name, separated by two underscores: the variable \( R_A \) with the instance index 27 would thus be renamed to \( R_A_{--27} \).

This phase only affects the AST and will thus not influence the Program Slicing. However since the BIL representation of single instructions will be shown in the GUI, the results of the SSA transformation should be visible.

### 3.6 Program Slicing

Most papers on Program Slicing discuss its application to higher level languages. The issues elaborated in those papers, e.g. preserving the syntax, are often specific to this domain and do not directly relate to our approach of performing Program Slicing on BIL. Some, like the paper “On Slicing Programs with Jump Statements” by H. Agrawal [8], are the helpful exceptions.

#### 3.6.1 Our Approach

Program slicing algorithms are generally based on data-flow information, which is inherently explicit in our setting, since we work on SSA code. In order to profit from the transformation to SSA, we do not implement one of the common algorithms. Our approach however resembles the idea of the algorithm laid out by M. Weiser [6].

Similar to his algorithm our backward slicing keeps track of a set of variables that potentially have an effect with respect to our slicing criterion. Both algorithms trace back from the point of interest and upon reaching the definition of one of its variables, extending that set by the variable uses in the definition. However, since definitions are inherently unique for variable instances in SSA, we can jump directly to the defining points of variables, instead of iterating over all statements and individually keeping track of the set of affecting variables.

More precisely we keep track of a set of affecting variable instances not variables, which implies that we can remove them from the set, once we have processed their definition (they cannot exist prior to their definition). Our algorithm terminates as
soon as all variables have been traced to their definition and the set is thus empty. By keeping track of the visited nodes to avoid loops and since we guarantee that every instance has a definition, by initializing all variables with ⊥ if no predecessor is present, the algorithm will always terminate.

We can also simplify the definition of the slicing criterion, due to the use of SSA. When multiple definitions of a variable are possible, only the combination of variable and point of interest allows a precise definition of a slice. Variable instances in SSA code, however have only one definition, removing the need for the point of interest as part of the slicing criterion. Our simplified slicing criterion is just a variable instance.

Algorithm 2 is the resulting algorithm for Program Slicing on SSA. It does however only return what M. Weiser denotes as $S[0]$ slice, a slice of all statements the directly affect the value of the slicing criterion. Other variables that influence the control-flow might indirectly also affect the value of the slicing criterion, compare Figure 3.5.

![Algorithm 2](image)

3.6.2 Working with Slices

The created slices are displayed in the user interface by highlighting the statements belonging to the slice in the control-flow graph. This has the advantage that the slice is always viewed in the context of the whole program, allowing the software engineer to easily relate the gained information to other knowledge over the program. One can also quickly switch between different slices without having to reorientate oneself in the code. Furthermore, displaying the whole control-flow graph significantly reduces the need to include control-flow statements into the slice.

Program slicing is triggered by selecting a variable instance as slicing criterion. The tool then computes both the backward and the forward slice and highlights them in
1: READ($X_1$)
2: if $X_1 < 1$ then
3: $Z_1 \leftarrow 1$
4: else
5: $Z_2 \leftarrow 2$
6: end if
7: $Z_3 \leftarrow \phi(Z_1, Z_2)$
8: WRITE($Z_3$)

Figure 3.5: The direct backward slice $S[0]$ for $Z_3$ would only include lines 3 and 5. Since $X_1$ decides which of the lines is executed, it affects the value of $Z_3$ indirectly. Based on an example by M. Weiser (1981) [6]

the CFG in different colors. Statements that belong to both slices are highlighted as statements of the backward slice, which is usually of greater interest.

3.6.3 Example slice

Using the resulting implementation, a slice can now help understanding the task dispatcher example.

At the end of block #3 the instruction `jsr Y[0x0000]` calls a subroutine at the address given in the Y-register. Since this is the only subroutine call with relative addressing in this code fragment, this is likely the execution of a task. Slicing can help us find out what determines the call target, hence the executed task. To construct the slice, we select the variable instance $R_{IY__3}$ as the slicing criterion. This is the instance of the register variable $R_{IY}$ used in the `jsr` statement. The generated slice is displayed on instruction level in Figure 3.6.

The first statement of the backward slice, loads register B from a memory location. Afterwards register A is cleared and a left shift is performed on register D. Here the BIL representation conveys information that is not obvious to the reader: Register D is composed of the two 16 bit registers A and B, in order to perform 32 bit operations. The value of the register is thus doubled and subsequently added with 0x2067. Afterwards `xgdy` swaps the value of register D and Y. The last statement of the slice loads from the memory location stored in Y and stores the value back to Y. This additional lookup indicates that a lookup table for the position of the task subroutines is stored at memory location 0x2067.

The actual jump is part of the forward slice, because it just reads the slicing criterion and does not write to the Y register. The fact that no other statements are contained in the forward slice allows us to conclude that this particular value of Y is not used anymore in the analyzed code.
Figure 3.6: Slice of the task dispatcher example. The slicing criterion is the instance of variable R_IY that is used in jsr Y[0x0000], at the end of block #3.
4 Evaluation

Goal of this bachelor thesis is the implementation of a tool that performs Program Slicing on binary intermediate language using static single assignment form. This chapter is an evaluation if the goal has been reached and how well the implementation performs.

During the SSA transformation, the tool performs data-flow analysis on the loaded BIL code and states the def-use-chains explicitly. The results of the data-flow analysis are then used for the subsequent Program Slicing.

The SSA transformation as such produces good results, as the implemented algorithm yields minimal form for all programs with a reducible CFG. Benchmarking the implementation, however, is difficult, as no directly comparable tool exists. The Binary Analysis Platform BAP contains a tool for SSA transformation of BIL, however due to our BIL extensions, both implementations would likely construct a different CFG for the same BIL input and would thus naturally produce different SSA form.

For a possible comparison, the following benchmarks are of interest:

**Minimality** Which implementation yields the least $\phi$-functions while producing a correct solution?

**Complexity** How long does each implementation take to perform the SSA transformation on the same input and how much memory is required?

4.1 Minimality

In order to gain some insights on how the tool performs, it was applied to a set of functions, taken from programs running on an embedded system.

The tested functions are listed in Figure 4.1 with their respective size, which can easily be determined from the BIL file. The number of CFG blocks and variables is determined during the CFG construction and the number of $\phi$-functions is counted as they are inserted during the RC-Approach and removed during the minimization (the table contains the number of $\phi$-functions remaining after the minimization).

The rather small *init* function is split into two blocks, because it contains a subroutine jump. While this break is not strictly necessary, it is inserted by the implementation as part of the jump and call handling and gives some insights in the SSA
implementation here. Since this function contains 14 variables, the RC-Approach of SSA will introduces 28 $\phi$-functions. The first 14 of them are placed in the beginning of block 1, where they are used to initialize the variables and do not have any arguments (compare Section 3.5.1), the rest is placed in the beginning of block 2, as set by the RC-Approach. During the minimization, those 14 $\phi$-functions in the second block are removed as the block has just one predecessor.

One expectation about the RC-Approach is that it inserts $\phi$-functions for all variables at the beginning of every CFG block. Hence the number of $\phi$-functions should match the number of CFG blocks times the number of variables. This expectation is confirmed in the available data.

Secondly, the data also confirms that minimization of some form is performed. For most functions, more than 60% of the $\phi$-functions introduced during the RC-Phase were later removed in the minimization. Due to a lack of comparison, it is hard to rate the result of the minimization.

### 4.2 Runtime Complexity

#### 4.2.1 SSA

The SSA transformation is performed in three subsequent phases, hence we can compute the runtime complexity individually for each phase.

**RC-Approach** To help determine the runtime complexity of this phase, all loops of the implementation are shown in Figure 4.2.

We use $m$ to denote the number CFG blocks, $v$ for the number of variables and $N$ is the total number of BIL statements. In a worst case scenario, the number of CFG

<table>
<thead>
<tr>
<th>Function</th>
<th>No. of BIL statements</th>
<th>No. of CFG Blocks</th>
<th>No. of Variables</th>
<th>No. of $\phi$-functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dispatcher</td>
<td>297</td>
<td>5</td>
<td>23</td>
<td>115</td>
</tr>
<tr>
<td>talker</td>
<td>1,111</td>
<td>38</td>
<td>20</td>
<td>760</td>
</tr>
<tr>
<td>control</td>
<td>433</td>
<td>19</td>
<td>23</td>
<td>437</td>
</tr>
<tr>
<td>message</td>
<td>1,215</td>
<td>21</td>
<td>25</td>
<td>525</td>
</tr>
<tr>
<td>init</td>
<td>67</td>
<td>2</td>
<td>14</td>
<td>28</td>
</tr>
<tr>
<td>write</td>
<td>260</td>
<td>8</td>
<td>22</td>
<td>176</td>
</tr>
<tr>
<td>data copy</td>
<td>583</td>
<td>7</td>
<td>22</td>
<td>154</td>
</tr>
<tr>
<td>task loop</td>
<td>697</td>
<td>24</td>
<td>26</td>
<td>624</td>
</tr>
<tr>
<td>timer</td>
<td>519</td>
<td>13</td>
<td>24</td>
<td>312</td>
</tr>
</tbody>
</table>
for block ∈ CFG.blocks do 
    for variable ∈ CFG.variables do 
        insert φ-function 
        for predecessor ∈ block.predecessors do 
            add argument to φ-function 
        end for 
    end for 
    for instruction ∈ block do 
        perform linear SSA transformation 
    end for 
    for variable ∈ CFG.variables do 
        change index of last instance to block number 
    end for 
end for

Figure 4.2: RC-Phase of the SSA transformation, reduced to the loops

block predecessors is only bound by m. For the number of BIL statements inside a CFG block \( n_i \), it holds that \( \sum_{i=0}^{m} n_i = N \), since every statement is just contained in one block.

Hence the complexity of the RC-Approach is

\[
O \left( \sum_{i=0}^{m} (v \times m + n_i + v) \right) \\
\Rightarrow O \left( m^2 \times v + N + m \times v \right)
\]

The number of CFG blocks m is maximal, when each block consist of just one statement, hence the worst case assumption for m is N.

Under the assumption that the BIL code for each type of instruction has a fixed set of variables, hence the variable names do not depended on the operands or location in the program, the maximal total number of variables is constant for a certain BIL generator and not dependent on the program size. With that assumption, the worst case runtime complexity for this phase is \( O(N^2) \)

**Minimization**  The minimization is performed on every variable and iterates as long as \( \phi \)-functions can be removed. Since each variable can only have one \( \phi \)-function per CFG block, the minimization for a specific variable has a maximum of m iterations. Inside the iterations, the implementation loops through all instances of a variable. Since one BIL statement can only define one variable and hence only spawn one instance, the
number of instances of one variable $v_i$ summed up over all variables $v$ is no larger than $N$.

The worst case runtime complexity of this phase is thus
\[
O \left( \sum_{i=0}^{v} (m \times v_i) \right) \Rightarrow O(m \times N) \Rightarrow O(N^2)
\]

**Renaming** In the last phase, all variable references in the BIL code are renamed according to the information determined in the previous phases. This operation is carried out in linear time, with respect to the input size, hence the runtime complexity class is $O(N)$.

The overall worst case runtime complexity for the SSA transformation is thus $O(N^2)$.

### 4.2.2 Program Slicing

The tool contains functions for both, backward slicing and forward slicing. Both slicing methods utilize a work list of variables currently under observation and iterates until the work list is empty. However the iteration only processes each BIL statement once, afterwards the statement can not again contribute to the work list. This assures termination and the worst case runtime complexity of $O(N)$.

### 4.3 Memory Complexity

To determine the memory complexity, we follow the data structures (Section 3.2) constructed by the tool. A CFG container is only constructed once per input file and thus does not affect the memory complexity. CFG blocks and instructions just group the BIL statements, hence their number is limited by the number of statements in the input file, $n$.

Likewise, the number of variables can not exceed $n$, since only one variable can be defined per statement. In the worst case scenario of $n$ CFG blocks and $n$ variables, the RC-Approach will introduce $n \times n$ additional $\phi$-statements. The total number of statements $N$ is thus $N = n + n \times n$. Variable instances can be spawned by every statement, but only one per statement, hence their number is bound by $N$. The number of variable references is not directly proportional to the number of statements, as different statements can have a different number of variable references, but we can assume a constant upper bound for the number of references per statement, hence the total number of variable references is the same complexity class as $N$.

The graph structure of the control-flow graph is stored in link lists of the CFG block objects. For the worst case assumption of a complete graph, the number of links is
bound by $n \times n$. However in practice the number of links is significantly lower and the usage of link lists, instead of an adjacency matrix, allows to take advantage of that.

Summarizing, most parts of the data structure scale linearly with the input size, while statements, variable references and the graph representation scale quadratic in the worst case scenario. Hence to worst case assumption for the whole data structure is also quadratic scaling.
5 Future Work

5.1 Handling Memory Access

BIL has two different kinds of variables, one that represents registers or flags and thus contains fixed width integers or boolean values and another type that represents memory. The current implementation does not differentiate between those types. As a result, every slice that includes a statement, which reads from a memory variable, also includes all statements that write to that memory variable.

To avoid this major loss in precision, two approaches could be considered.

Exclude memory variables  By excluding memory variables from the slicing process, the precision of the resulting slice is restored, however the slice is significantly limited in its reach, because tracing would stop when a variable is defined from a memory load operation. The exclusion of variables could either be specified by the user or could automatically be determined during the CFG construction.

Variable detection  A much better result could probably be accomplished by tracing values inside memory variables. This should be simple for directly addressed memory access, however indirect addressing will require heuristics or supporting data flow analyses to determine possible access targets as well as a memory representation and SSA implementation that can handle imprecise memory access.

5.2 Liveness Analysis for SSA Minimization

During the minimization phase of the SSA algorithm, $\phi$-functions that just perform variable renaming or combine inherently equal predecessors are removed. The output should match the result of the dominance frontier based algorithm by R. Cytron et al. [4] and the common definition of minimal SSA form for reducible CFGs.

The number of $\phi$-functions could potentially be further reduced by considering the liveness of a variable instance. $\phi$-functions that define a variable instance, which is never used, could be removed.
5.3 Slicing on Multiple Variable Instances

The original definition by M. Weiser [6] states that a slicing criterion has the form \((i, \{x\})\) with a set of variables \(\{x\}\) and a point of interest \(i\). In the explanation of our Program Slicing approach (section 3.6.1) we argued that variable instances in SSA code eliminate the need for a specified point of interest. While this does not pose a restriction, we also reduced the slicing criterion to just one variable instance, reducing its expressiveness.

Our algorithm (see Algorithm 2) could be modified to take a set of variable instances as slicing criterion. This would extend the expressiveness of the slicing criterion even beyond the original definition, because variable instances from different positions inside the code could be part of the criterion, instead of a single point of interest.

To make this feature accessible, the user interface would have to be extended.

5.4 Control-Flow-Statements

The current Program Slicing implementation constructs a slice of all the variable assignments that affect the value of the slicing criterion. Control-flow statements, however, are not part of the slice, even though the variables used in conditional statements can indirectly affect the slicing criterion.

In a first step, the above extension of the slicing criterion would allow the user to manually include variables that are used in conditional statements.

In a second step, this could become part of the automatic slicing algorithm. This should be considered carefully however. In their algorithms, M. Weiser [6] and S. Horwitz et al. [9] use dominance information about the CFG to determine, which control-flow statements should be included. Our SSA implementation is specifically build on an algorithm that does not require dominance information. If dominance information would be included, the selection of the SSA algorithm should be questioned as well, as other algorithms that could reuse this information would likely be more efficient.

From a concept point of view, this extension of the slice would significantly change the workflow. A slice would likely contain a complete program behavior, not just the data-flow information it currently conveys. This could reduce readability of the information, as much larger portions of the code would be highlighted. To alleviate the effect, one could highlight the statements in multiple shades, representing how directly a statement influences the slicing criterion.
5.5 Inter-Procedural Slicing

A major limitation of the current implementation is the intra-procedural slicing. Almost all programs use function calls and the lack of handling the results and invalidated registers is a huge loss of precision.

Addressing this by function inlining would result in an exponential blow-up of the CFG and recursive calls could not be represented. Constructing an inter-procedural CFG or supergraph would be more efficient, at the price of loosing a lot of precision. A more sophisticated and possibly more specialized solution would be desirable.
6 Conclusion

Over the course of this bachelor thesis, a tool has been implemented that assists software engineers in the understanding of machine code. It does so by performing Program Slicing on platform independent binary intermediate language (BIL), using a transformation to Static Single-Assignment form (SSA).

The application domain and the chosen methods yielded a few interesting and - to the best of my knowledge - new combinations. Program slicing is commonly applied on structured high level languages, here it’s applied to machine code. The slice is displayed on assembler level, however the slicing is performed on the low-level, platform independent BIL representation.

Using BIL as input format was a good choice. It enables the desired platform independence, allows flexible annotations to be used to influence the proceeding of the algorithm, and contains information that allow working on instruction level and hiding the low level information, when necessary. Hence it has met the expectations.

The SSA transformation can be performed on BIL easily and during it’s construction, the information required for Program Slicing is gained in such quality that the level of slicing performed by our tool is largely simplified. Actually transforming the BIL code to SSA form helps the user when following a slice on BIL level and when determining a slicing criterion, although it is not strictly required for the actual slicing. The simple SSA algorithm eliminated the need to calculate dominance information about the CFG. However future work directed at improving the slicing might require dominance information for the slicing algorithm, in which case the SSA transformation might also benefit from a switch to an algorithm based on dominance information.

Developing in C# was a good experience and the functional language features helped simplify the code. Describing the WPF based user interface in XML simplified the data binding.

The generates slices can help in the understanding of programs by both excluding statements that are irrelevant and hinting those which are relevant but not obviously recognizable on assembler level. The ideas given for future work may even improve the effectiveness of slicing and lead to more complete slices, reducing the manual work further.
Bibliography


