INTERPROCEDURAL STATIC SINGLE ASSIGNMENT FORM

by

Silvian Calman

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2011 by Silvian Calman
Abstract

Interprocedural Static Single Assignment Form

Silvian Calman

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2011

Static Single Assignment (SSA) is an Intermediate Representation (IR) that simplifies the design and implementation of analyses and optimizations. While intraprocedural SSA is ubiquitous in modern compilers, the use of interprocedural SSA (ISSA), although seemingly a natural extension, is limited. In this dissertation, we propose new techniques to construct and integrate ISSA into modern compilers and evaluate the benefit of using ISSA form.

First, we present an algorithm that converts the IR into ISSA form by introducing new instructions. To our knowledge, this is the first IR-based ISSA proposed in the literature. Moreover, in comparison to previous work we increase the number of SSA variables, extend the scope of definitions to the whole program, and perform interprocedural copy propagation.

Next, we propose an out-of-ISSA translation that simplifies the integration of ISSA form into a compiler. Our out-of-ISSA translation algorithm enables us to leverage ISSA to improve performance without having to update every compiler pass. Moreover, we demonstrate the benefit of ISSA for a number of compiler optimizations.

Finally, we present an ISSA-based interprocedural induction variable analysis. Our implementation introduces only a few changes to the SSA-based implementation while enabling us to identify considerably more induction variables and compute more loop trip counts.
Acknowledgements

First, I would like to thank my advisor, Professor Zhu. Without his suggestions and guidance, this work could not have progressed as well as it has. I am also very fortunate and grateful that Professor Abdelrahman and Professor Steffan are on my committee. I would like to thank them for their help and insight. In addition, I would also like to thank Professor Amaral for serving on my committee.

I am very thankful to my family who helped me tremendously throughout the course of all my studies. The support and encouragement, from my parents especially, have enabled me to complete this work.

Finally, I thank Hana, for her constant love, help, and encouragement.
# Contents

1 Introduction  
  1.1 State Of The Art .................................................. 5  
  1.2 Contributions ...................................................... 7  
  1.3 Thesis Overview ................................................... 9  

2 Background and Related Work  
  2.1 Introduction ....................................................... 10  
  2.2 Notation ............................................................. 11  
  2.3 Intermediate Representation ...................................... 11  
  2.4 Relevant Interprocedural Analyses ................................. 16  
  2.5 Overview of Static Single Assignment ............................ 17  
    2.5.1 SSA Variables .................................................. 20  
  2.6 Static Single Assignment Extensions ............................... 21  
    2.6.1 Static Single Assignment Extensions That Support Aliased Variables 22  
    2.6.2 Array Static Single Assignment ................................. 24  
    2.6.3 Prior Work on Interprocedural Static Single Assignment ............ 25  

3 Interprocedural Static Single Assignment Form  
  3.1 Introduction ....................................................... 27  
  3.2 IR Extensions ...................................................... 28  
    3.2.1 Handling Aliased Program Variables .......................... 28
3.2.2 Passing Values Across Procedures .................................. 29
3.3 Interprocedural Static Single Assignment Example ................. 30
3.4 Copy Propagated ISSA ....................................................... 32
  3.4.1 Folding $\phi^C$ Instructions ........................................ 33
  3.4.2 Example and Terminology ........................................... 34
3.5 Comparison .................................................................. 39

4 Interprocedural Static Single Assignment Construction ............. 41
  4.1 Introduction ................................................................. 41
  4.2 Pointer Analysis ............................................................ 44
  4.3 Choosing SSA Variables .................................................. 45
    4.3.1 Invocation Count Analysis ......................................... 45
    4.3.2 Heap Allocation Sites Executed Once At Most ................. 47
  4.4 Dereference Conversion ................................................ 47
  4.5 Inserting $\phi^V$ and $\phi^C$ Instructions .............................. 50
    4.5.1 Procedure Mod/Ref Analysis ....................................... 50
    4.5.2 ISSA Liveness Analysis ............................................ 52
    4.5.3 Pruning $R\mathcal{E}F$ and $MOD$ .................................. 57
    4.5.4 Inserting $\phi^V$ and $\phi^C$ Instructions ...................... 58
  4.6 Interprocedural Copy Propagation .................................... 59
    4.6.1 Algorithm to Fold $\phi^V$ and $\phi^C$ Instructions ............ 61
  4.7 Interprocedural Value Replacement ................................... 65
    4.7.1 Testing Interprocedural Value Equality ......................... 66
    4.7.2 Computing the Interprocedural Value Replacement Map ....... 69
  4.8 Experimental Evaluation ............................................... 72
    4.8.1 Scalability ............................................................. 74
    4.8.2 Excluded Benchmarks .............................................. 76
    4.8.3 Impact of Increasing the Scope and Resolution ............... 76
4.8.4 Impact of Copy Propagation and ISSA Liveness Analysis . . . . 78
4.8.5 Impact of Pointer Analysis . . . . . . . . . . . . . . . . . . . . . 79
4.8.6 ISSA IR in Benchmarks . . . . . . . . . . . . . . . . . . . . . . 81
4.8.7 Library Calls . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
4.9 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

5 Out-of-ISSA Translation . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.2 Background and Related Work . . . . . . . . . . . . . . . . . . . . . . . 86
5.2.1 Out-of-SSA Translation . . . . . . . . . . . . . . . . . . . . . . . 86
5.2.2 Challenges and Opportunities of Out-of-ISSA Translation . . . . 88
5.2.3 Running Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.3 Applied Compiler Passes . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.3.1 Storage-Remap Transformation . . . . . . . . . . . . . . . . . . . 91
5.3.2 Applied Passes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.4 Proposed Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.4.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.4.2 Simplifications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.4.3 Library Calls . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.4.4 Framework . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
5.4.5 Selecting the Propagation Variable . . . . . . . . . . . . . . . . . 102
5.4.6 Committing Propagation Variables . . . . . . . . . . . . . . . . . 110
5.4.7 Converting ISSA Form Back to SSA Form . . . . . . . . . . . . . 114
5.4.8 Impact of Transformations and Out-of-ISSA Translation . . . . .115
5.5 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
5.5.1 Constant Propagation . . . . . . . . . . . . . . . . . . . . . . . . . 121
5.5.2 Dead Code Removal . . . . . . . . . . . . . . . . . . . . . . . . . 121
6 ISSA-Based Interprocedural Induction Variable Analysis

6.1 Introduction .............................................. 125
6.2 Background and Related Work .......................... 127
6.3 Motivation ................................................. 130
6.4 Baseline Induction Variable Analysis ................. 132
   6.4.1 LLVM Scalar Evolution Pass ..................... 132
   6.4.2 Monotonic Induction Variables .................. 134
6.5 ISSA-Based Interprocedural Induction Variable Analysis ............................................. 137
   6.5.1 Predicated Induction Variables ................. 138
   6.5.2 Supporting ISSA Form ............................ 139
   6.5.3 Context Sensitivity ............................... 140
   6.5.4 File-Position Induction Variable Analysis .... 144
6.6 Case Studies .............................................. 149
   6.6.1 Scalar Globals ................................... 149
   6.6.2 Induction Variables with Interprocedural Recurrence Relations 151
   6.6.3 Heap-Allocated Structure Fields ............... 152
   6.6.4 File-Position Induction Variables ............. 153
6.7 Experimental Evaluation .................................. 155
6.8 Summary .................................................. 159

7 Conclusions and Future Work .......................... 160

7.1 Constructing ISSA Form ................................ 160
7.2 Out-of-ISSA Translation ................................ 161
7.3 ISSA-Based Interprocedural Induction Variable Analysis ............................................. 162
7.4 Limitations ............................................... 162
7.5 Future Work ................................................................. 163
    7.5.1 Reducing the Cost of ISSA Construction ...................... 163
    7.5.2 Propagating Values Across Procedures ......................... 164
    7.5.3 Applications of ISSA ............................................. 165
7.6 Closing Remarks ....................................................... 166

Bibliography .............................................................. 167
List of Tables

4.1 Benchmark characteristics and the time it takes to construct ISSA (column labeled Time), in seconds. 73
4.2 Number of SSA variables, load instructions replaced, and singular allocation sites identified. 77
4.3 Impact of ISSA liveness analysis and copy propagation measured by the reduction in the number of $\phi^V$ and $\phi^C$ instructions. 79
4.4 Size of $\text{REF}$ and $\text{MOD}$ when generating ISSA with a field-insensitive and field-sensitive pointer analysis. 80
4.5 Numerical summary of the data in Figure 4.11, which includes the percentage and relative space consumption of the new ISSA instructions. 81
4.6 The number of load and store instructions inserted to write and retrieve the value of SSA variables around call instructions invoking library procedures. 84
5.1 Compilation and program runtime in seconds. The program runtime is provided for the LLVM baseline ($\text{LLVM}$ column), the LLVM baseline with the passes described in this report ($\text{LLVM+}$ column), and the LLVM baseline with the out-of-SSA algorithm adapted for ISSA IR where globals are used instead of locals ($\text{Adapt}$ column). In the Speedup Factor columns, we provide the program performance improvement for the IR generated using the two out-of-ISSA translation algorithms against the LLVM baseline (i.e. divide program runtime numbers). 118
5.2 Number of arithmetic, pointer arithmetic, and branch instructions (as well as aggregate improvements) constant folded using our algorithm over the LLVM constant propagation (-instcombine, -ipconstprop) .................................. 122

5.3 Number of basic blocks left after baseline passes are applied (column labelled \textit{LLVM}) and after LLVM passes along with our proposed passes are applied on ISSA form (column labelled \textit{LLVM+}) .................................................. 123

5.4 The number of instructions removed when running the common subexpression elimination pass on SSA (column labelled \textit{SSA}) and ISSA (column labelled \textit{ISSA}) form. When run on ISSA form, the common subexpression elimination pass removes 42.9\% more instructions. ................................. 124

6.1 The time it takes to perform the induction variable analysis in millisecond. 155

6.2 The number of induction variables found. Columns labeled \textit{LLVM} contain the baseline numbers. We differentiate between induction variables found by tracing the recurrence relation interprocedurally in the columns labeled \textit{Inter} (with) and \textit{Intra} (without) .................................................. 156

6.3 The number of loop trip counts computed. Columns labeled \textit{LLVM} contain the baseline numbers. The \textit{ISSA} column contains the number of trip counts found due to ISSA form alone, while the \textit{ISSA+IV} column also considers the newly discovered induction variables when computing the trip count. .................................................. 157

6.4 Performance of the file-position induction variable analysis. We differentiate between induction variables found when tracing the recurrence relation interprocedurally in the columns labeled \textit{Inter} (with) and \textit{Intra} (without). 159
List of Figures

1.1 Def-use chain example taken from Allen and Kennedy [3].................. 2

1.2 Program specialization using ISSA. In this example, P1 and P2 are sections of code, on some execution path following the call to procedure init. 5

2.1 SSA form example. ...................................................... 19

2.2 Illustration of may def-use relations that occur when a single virtual SSA variable represents multiple program variables (y and z). In this scenario, load and store instructions whose pointer value is @var can access either y or z. When replacing uses of var with a single definition during ISSA construction, we create may def-use relations as we are not certain which program variable (either y or z) is accessed. ...................... 22

3.1 Interprocedural SSA example. ........................................... 31

3.2 Example to illustrate the folding of φC instructions based on the usage point. ................................................................. 33

3.3 Example to illustrate the folding of φC instructions in SCCs. .......... 35

3.4 C source code and its associated SSA form. Note that %v5 in Figure 3.4, line 17, is either equal to @z or @y, hence its dereferences (on line 20 and line 23) can access the variables y and z. .............................. 36

3.5 ISSA form for the code shown in Figure 3.4. ............................. 38
4.1 Overall procedure for ISSA generation, which is outlined in Section 4.1. Details are provided in the rest of this chapter. 

4.2 Example illustrating a scenario where the flow-sensitive interprocedural liveness analysis computes a more precise result than the ISSA liveness analysis. In this example, a flow-sensitive interprocedural liveness analysis can determine that $a$, $b$, and $c$ do not have to be propagated into procedure $proc$ whereas the ISSA liveness analysis cannot.

4.3 Demonstrating why folding $\phi^V$ instructions merging a single value (prior to folding $\phi^C$ instructions) is legal.

4.4 Example to illustrate the replacement of $\phi^C$ instructions using Algorithm 4.8. At the entry to $BB_2$, we set $VirtVal[Y]$ and $VirtVal[Z]$ to $\emptyset$ because $\mathcal{VTD}[BB_2] = \{Y, Z\}$.

4.5 Demonstration of how call graph paths can be leveraged to determine whether we can replace one instruction with another. In $BB_1$ and $BB_3$, the path on the left corresponds to procedure $foo_3$. As shown in the figure above, the path to the last invocation of $foo_3$ after the addition instruction defining $%x^2$ is $\langle CI_2, CI_3 \rangle$.

4.6 Top-down propagation of the virtual SSA variable values.

4.7 Examples illustrating how $\mathcal{IVR}$ derives relevant call graph paths from call sites.

4.8 Ending call site graph for procedure $foo_3$ in the example from Figure 4.5.

4.9 Number of $\phi$ instructions inserted in the benchmarks $255.vortex$ and $176.gcc$ as we process methods during $\phi$ placement.

4.10 The number of store and load instructions where the pointer value can access $1\rightarrow3$, $3\rightarrow10$, $10\rightarrow50$, and over 50 SSA variables.
4.11 The percentage of $\phi^L$, $\phi^S$, $\phi^V$, $\phi^C$, and $\phi$ instructions as well as the space they occupy. Space consumption is computed by adding the number of instructions and their operands. ........................................... 82

4.12 The percentage of $\phi^L$, $\phi^S$, $\phi^V$, $\phi^C$, and $\phi$ instructions and the memory space they occupy in relation to all other instructions. ......................... 83

5.1 Example illustrating translation out of SSA form. ......................... 87

5.2 Example to illustrate ISSA form and out-of-ISSA translation. ........... 90

5.2 Example to illustrate ISSA form and out-of-ISSA translation (continued). 92

5.3 Overall procedure for out-of-ISSA translation, which is outlined in Section 5.4.1. Details are provided in the rest of Section 5.4. ................. 96

5.4 In Figure 5.4(a) we illustrate the distribution of SSA variables into scalar globals, non-scalar globals, and stack and heap allocated variables. In Figure 5.4(b), we illustrate the percentage decrease in the number of arguments that occurs when we use our passes (“LLVM+”) in addition to the LLVM baseline (“LLVM”). .................................................. 119

5.4 In the subfigures above, we illustrate the percentage decrease in the number of store instructions and load instructions that occurs when we use our passes (“LLVM+”) in addition to the LLVM baseline (“LLVM”). (Continued) ........................................................................ 120

6.1 Example illustrating the induction variable analysis. ....................... 127

6.2 Motivating example for ISSA-based induction variable analysis. ....... 131

6.3 Example based on the C source code in Figure 6.2 that illustrates the file-position analysis. ................................................................. 148

6.4 C source code fragment illustrating a loop in the benchmark 300.twolf in SPECINT2000 [1] where a global variable (row) is a linear induction variable. ................................................................. 150
6.5 Example illustrating an induction variable found by computing the recurrence relation across procedures. It is taken from the benchmark 197.parser in SPECINT2000 [1] (files xalloc.c and main.c). . . . . . . . . . . . . . . . 151

6.6 Example illustrating a heap-allocated induction variable in the benchmark JPEG in MediaBench [31] (Relevant source code can be found in the files jcmaster.c and jcmaintct.c). . . . . . . . . . . . . . . . . . . . . . . . . . . 152

6.7 C source code that reads from a file. It is located in the benchmark GSM in MediaBench (in main.c). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
List of Acronyms

IR  Intermediate Representation

SSA  Static Single Assignment

ISSA  Interprocedural Static Single Assignment

LLVM  Low Level Virtual Machine

SCC  Strongly Connected Component

BDD  Binary Decision Diagram

CFG  Control Flow Graph
Chapter 1

Introduction

The compiler translates programs written in a programming language to machine code that can then be executed by a computer. Typical compilers [2] support high-level programming languages, target multiple architectures, identify and report certain programmer errors, and improve the performance of the target program. This is accomplished by applying a number of compiler optimizations, which transform the code in order to obtain various improvements.

While it is tempting to believe that the compiler’s responsibility ends with the production of correct machine code, this function alone is not enough [3]. Performance is of paramount importance in computer systems and compilers are expected to generate suitably efficient programs for such systems. The optimizing compiler improves programmer productivity, because it allows developers to focus on improving software maintainability and reducing the “time to market” rather than on tuning code for optimal performance and subsequently debugging the modifications. This reduces the costs associated with software development, which in turn makes software more affordable. Since the success of many innovations in computer architecture and programming languages is contingent on the ability of the compiler to harness their potential, many hardware and software vendors actively develop and maintain optimizing compilers.
Associating the definitions and uses of program variables is a fundamental step for a large number of compiler optimizations. To accomplish this, a compiler can compute def-use chains. For a given variable, the def-use chains will identify the uses reached by any definition. We say that a definition $D$, of variable $v$, reaches one of its uses, $U$, if $v$ may be last defined by $D$ when executing instruction $U$.

While useful, the def-use chains have a number of drawbacks. First, def-use chains are usually discarded once a compiler pass is finished, since maintaining them while applying transformations complicates the implementation. Second, the number of def-use edges in a graph representing the def-use chains can grow very large in the presence of control flow [3]. Consider Figure 1.1(a), since control flow merges at node $S_4$, then each definition of variable $X$ can reach every use. Hence, we will have nine def-use edges: $\langle S_1, S_5 \rangle$, $\langle S_1, S_6 \rangle$, $\langle S_1, S_7 \rangle$, $\langle S_2, S_5 \rangle$, $\langle S_2, S_6 \rangle$, $\langle S_2, S_7 \rangle$, $\langle S_3, S_5 \rangle$, $\langle S_3, S_6 \rangle$, $\langle S_3, S_7 \rangle$. Moreover, a number of compiler optimizations, such as constant propagation [49, 50], rely on identifying basic blocks whose predecessors have different reaching definitions. One way to reduce the number of def-use edges is to kill variable definitions at such basic blocks. For instance, if we insert the assignment $X = X$ at the entry to $S_4$, as shown in Figure 1.1(b), then a single definition reaches each (original) use of $X$ and we reduce the number of
def-use edges from nine to six \( (\langle S_1, S_4 \rangle, \langle S_2, S_4 \rangle, \langle S_3, S_4 \rangle, \langle S_4, S_5 \rangle, \langle S_4, S_6 \rangle, \langle S_4, S_7 \rangle) \).

Leveraging this observation, *Static Single Assignment* (SSA) was proposed in the late 1980s [4, 17, 41] to address the drawbacks of def-use chains. SSA is an *Intermediate Representation* (IR) of the program, constructed for a set of program variables, which we refer to as *SSA variables*. In SSA form, each assignment to an SSA variable \( \text{var} \) creates a unique temporary that holds the value of \( \text{var} \). For instance, in the SSA form for Figure 1.1(a), shown in Figure 1.1(c), the temporaries \%v_1, \%v_2, and \%v_3 are created at definitions of \( X \). Furthermore, the IR is extended with a \( \phi \) instruction, which is inserted at control flow joins to merge temporaries created at different assignments of the same SSA variable. For instance, the \( \phi \) instruction in node \( S_4 \) of Figure 1.1(c) selects between the temporaries \%v_1, \%v_2, and \%v_3 based on the incoming edge and becomes the only reaching definition of \( X \) at all its uses. Algorithms that construct SSA form [18] and translate out of SSA\(^1\) [11, 18, 44] have been proposed and are widely used.

SSA form simplifies compiler analyses and optimizations since def-use chains are explicit in SSA\(^2\). Furthermore, SSA form enables fast, flow-insensitive algorithms to achieve many of the benefits of flow-sensitive algorithms without expensive data-flow analyses [30]. Due to these benefits, many modern compilers use SSA form. For instance, in order to simplify the design and implementation of transformations and optimizations, GCC [26] added an SSA form based optimization package named tree-SSA [35], while LLVM [30] adopted SSA form from the very beginning.

Many compiler optimizations are confined to the scope of a single procedure. That is, they are *intraprocedural*. However, modern programs can contain a large number of procedures. In order to optimize such programs, it is important to apply compiler optimizations across procedure boundaries. Modern compilers use two techniques to accomplish this. The first is *inlining*, which replaces call instructions with the body of

---

\(^1\)The out-of-SSA translation converts the IR from SSA to standard form.

\(^2\)Each SSA variable use is replaced with a temporary that corresponds to the *single* reaching definition.
the called procedure, allowing the compiler to apply intraprocedural optimizations on code that was originally located within different procedures. This technique is useful but compilers often limit the amount of inlining in order to restrict code size growth. The second technique is to enhance intraprocedural compiler optimizations in the presence of call instructions and pointers by leveraging interprocedural data-flow analyses; which are techniques for compile-time reasoning about the run-time flow of values. For example, side-effects analysis can be used to identify the set of variables written at a call site and hoist code out of loops that contain procedure calls. While useful for a number of applications, interprocedural data-flow analyses can be computationally expensive and will typically compute specific information that is useful only to a single optimization.

One way to address these problems is to build upon the success of SSA and extend the scope of definitions to the whole program. This extension is commonly referred to as Interprocedural SSA (ISSA). Naturally, it can be expected that ISSA can ameliorate the benefits derived from SSA-based analyses and optimizations without requiring additional interprocedural data-flow analyses.

Furthermore, the explicit identification of the program-wide uses of a definition in ISSA enables us to quickly evaluate the impact of interprocedural optimizations and simplify program transformations. For instance, we can leverage ISSA to create multiple versions of a given section of code, which are optimized for a given value of a variable. To illustrate this concept, we note that the global variable $TS$ is used by two branch instructions in Figure 1.2(a). Hence, we can duplicate the section of the program where the global variable $TS$ is used, such that it is executed only when $TS$ is equal to 0. In Figure 1.2(b), we present the resulting program, where two branch instructions are removed as a result of this transformation. In addition to folding instructions, this transformation can be used to enable and improve a number of compiler optimizations. For instance, loop unrolling and auto-vectorization will be more effective in the new version of the loop, since the variable $loopI$ is incremented by 1 on every iteration and
Figure 1.2: Program specialization using ISSA. In this example, $P_1$ and $P_2$ are sections of code, on some execution path following the call to procedure $\text{init}$.

The trip count\(^3\) is 16 (a constant).

The ISSA form of the program in Figure 1.2(a) can be used to identify such an optimization opportunity, because we would be able to observe that a definition of $TS$ is used by many branches. Moreover, since the program-wide uses of a definition are explicit, ISSA form simplifies copy propagation in the newly created version and can also be leveraged to perform value inference. Hence, ISSA can be used to identify interprocedural optimization opportunities as well as simplify interprocedural transformations.

### 1.1 State Of The Art

Although seemingly a natural extension, to date the use of ISSA in compilers is limited. One drawback is the cost of constructing ISSA form and the lack of demonstrated benefits.

---

\(^3\)The trip count of loop $L$ is the number of trips through $L$ prior to exiting it.
to compiler analyses and optimizations. Furthermore, in order to integrate ISSA form into a compiler we need to either update every compiler pass, which is an expensive and time consuming process, or convert the IR to SSA form using an \textit{out-of-ISSA translation}, so that we can leverage compiler passes that were not updated. While an out-of-ISSA translation simplifies the integration of ISSA form in compilers, maintaining the performance of the resulting code is a challenge. Given the great potential of ISSA form, it seems intuitive that a comprehensive study on it would already have been completed. However, this is not the case. Prior to our work, the tradeoff between the benefit and cost of ISSA form was never thoroughly evaluated in the literature.

In fact, to the best of our knowledge, only two ISSA construction algorithms were published. Liao [32] applied unification-based pointer analysis (Steensgaard’s [46]) and renamed memory accesses to their corresponding alias set. Staiger et al. [45] used symbolic variables called locators to represent aliased program variables within each procedure. In this work, values are passed interprocedurally by mapping locators onto one another and SSA is generated in a traditional way [18] by utilizing the point-to graph to map pointer dereferences to their corresponding locator. Staiger et al. [45] showed that an inclusion-based pointer analysis (Andersen’s [5]) reduces memory consumption and considerably speeds up the formation of ISSA, compared to unification-based pointer analysis (Steensgaard). While Staiger et al. [45] evaluated the memory consumption and the construction time, ISSA is a data structure in both of these algorithms rather than an IR. Moreover, neither Liao nor Staiger evaluate the impact of ISSA on common compiler analyses and optimizations. In fact, only Liao leveraged ISSA for a client application (the demand-driven slice computation algorithm).
1.2 Contributions

While ISSA clearly enhances a large number of compiler analyses and optimizations, there are four important questions that prior research cannot answer.

1. What is the cost of constructing ISSA form and increasing the scope of definitions to the whole program?

2. How can we translate out of ISSA form without degrading program performance?

3. Can a production compiler with legacy passes that were built on SSA be adapted to generate high-performance code on ISSA form?

4. What is the impact of ISSA form on compiler analyses and optimizations?

By addressing these concerns, we can identify the key problems and their impact when constructing and using ISSA form, thus generating a solid foundation for additional research. Using this study, future research can determine how the construction of ISSA form should be modified in order to enable new applications, enhance current results, and derive the same benefits at a lower cost.

This dissertation focuses on integrating ISSA form into a compiler and its benefit to client applications. At a high level, it makes three contributions:

1. We propose an ISSA construction algorithm that improves on previous work in a number of ways. First, we use a field-sensitive pointer analysis, which significantly reduces the number of instructions we insert and enables us to include structure fields in the set of SSA variables. Second, in addition to structure fields, we also include certain heap and stack allocated variables in the set of SSA variables. Third, in order to improve the efficiency of ISSA construction, we propose an interprocedural live variable and undefined variable analysis that reduces the input and output instructions that would have been inserted by 24.8%. Finally, we propose an interprocedural copy propagation algorithm that removes an additional 44.5% of the
input and output instructions. We implemented our algorithm in the LLVM infrastructures [30] and validated our proposed techniques on a set of MediaBench [31] and SPECINT2000 [1] benchmarks.

2. We present an out-of-ISSA translation algorithm and a storage-remap transformation that enable us to integrate ISSA form into a compiler while generating efficient code. While the out-of-ISSA translation can be used to leverage ISSA form without updating every compiler pass, we observed that a naive extension of out-of-SSA translation generally degrades program performance. In contrast, our proposed algorithm and the storage-remap transformation improve program performance on a set of MediaBench [31] and SPECINT2000 [1] benchmarks by a factor of 1.02 when compared to the LLVM baseline [30]. This is due to the removal of a large number of store instructions, load instructions, and parameters as well as a set of compiler optimizations that leverage ISSA form.

3. We propose an ISSA-based interprocedural induction variable analysis and demonstrate that it significantly increases the number of induction variables found, as well as the number of constant and loop invariant trip counts that are computed. Algorithms found in the literature and implementations in free-source compilers such as GCC [26] and LLVM [30] rely on SSA form. However, the set of SSA variables is limited to scalar stack variables whose address is not taken. We describe how ISSA form can be leveraged to extend induction variable analysis interprocedurally to include the following: globals, singleton heap variables, structure fields, and files. We implemented our induction variable analysis and compared it against the LLVM infrastructure for a set of MediaBench [31] and SPECINT2000 [1] benchmarks. We observed an average increase of 14.4% and 49.1% in the number of polynomial and monotonic induction variables, respectively. Furthermore, using ISSA form and our induction variable analysis, we computed 1.1 times more constant trip counts and
2.6 times more loop invariant trip counts.

## 1.3 Thesis Overview

The remainder of this dissertation is organized as follows: Chapter 2 provides background information, introduces our IR, and reviews the evolution of SSA as well as its relevant extensions. Chapter 3 presents and motivates the ISSA form proposed by us, including key details regarding interprocedural copy propagation. Chapter 3 also defines key terminology used throughout this dissertation.

The main contributions of this work, as summarized above, are presented in Chapters 4, 5, and 6. In Chapter 4, we present our ISSA construction algorithm. In Chapter 5, we present the proposed out-of-ISSA translation algorithm and the storage-remap transformation. Chapter 6 contains our ISSA-based interprocedural induction variable analysis. To improve the presentation, each chapter contains both a description of the algorithms and techniques used, as well as their experimental evaluation.

Chapter 7 concludes this dissertation and discusses future research directions.
Chapter 2

Background and Related Work

2.1 Introduction

The conversion of source code written in a programming language to an executable is done in a number of steps [2]. First, the source program is usually passed to a preprocessor, which gathers modules from separate files and expands macros into source language statements. The modified program is then inputted to the compiler, which analyzes the source code and outputs assembly code that is then used by the assembler to produce relocatable machine code. The assembler outputs this code into object files or libraries. A linker will then merge object files and libraries into one file by resolving references to externally defined symbols.

The compiler accepts the program in the form of a character stream. The compiler then tokenizes the program during a lexical analysis and later parses it in order to build a syntax tree. In the next step, a semantic analysis leverages the derived syntax tree to check the source program for semantic consistency with the language definition [2]. Afterwards, compiler optimizations are applied on an Intermediate Representation (IR) of the program. In many compilers, such as GCC [26] and LLVM [26], the syntax tree (which is also an IR) is converted into three-address code, which is the IR used in the
optimizer and the code generator. In recent years, modern compilers adapted Static
Single Assignment (SSA) form as the IR in order to simplify the implementation of
compiler analyses and optimizations as well as reduce the resources they consume.

This chapter presents the background and related work that forms the basis of our
proposed research. In Section 2.2, we describe the notation for various concepts. In
Section 2.3, we provide an overview of the IR that we will use and explain various
concepts regarding the IR, which are used in this dissertation. In Section 2.4, we provide
background material on relevant interprocedural analyses. In Section 2.5, we review SSA,
covering its development, uses, and adaptation. In Section 2.6, we survey the literature
on relevant extensions of SSA.

2.2 Notation

This dissertation contains C programs, IR examples, and algorithms. When presenting
the IR of programs, examples, and algorithms we use the operator := as the definition
operator and = is the equality operator. In our notation, \( x := y \) denotes an assignment
from variable \( y \) to variable \( x \) and the operation \( x = y \) will evaluate to “true”, if \( x \) is equal
to \( y \) and to “false”, otherwise.

In this dissertation, we manipulate maps; square brackets are used to query and
update these maps. For instance, \( \mathcal{E}_X[k] := val \) will assign the value \( val \) to a key \( k \) in the
map \( \mathcal{E}_X \), while \( \mathcal{E}_X[k] \) will retrieve the value associated with \( k \). We represent a sequence
using tuple notation (i.e. \( \langle \rangle \)).

2.3 Intermediate Representation

In this section, we review the adopted IR and explain various concepts regarding the
IR that are used in the rest of this dissertation. A more comprehensive introduction
to IR is provided by Aho et al. [2]. The IR consists of data structures for procedures,
basic blocks, and instructions. Within the data structure for a procedure we maintain a sequence of basic blocks, each of which contains a sequence of instructions. The parent of an instruction will be the basic block in which it is contained and the parent of a basic block is the procedure in which it is located. Moreover, we refer to the procedure in which an instruction $I$ is located, as the *parent procedure* of $I$.

Now that we have discussed the structure and contents of the IR, we present the grammar:

```
instrn : [label ':'] stmt
stmt : [temporary ' := '] expr ';'
expr : opcode op_list
op_list : op | op_list ',' op
op : value | '( value , value )'
value : cnst | string | label | temporary | var | procedure
```

As it can be observed, the IR contains values in addition to instructions. A value is either:

- The address of a global variable or procedure ('@' prefix).
- A numeric constant ('#' prefix).
- A *named temporary* that holds the result of an instruction or the value of a parameter ('%' prefix). We refer to the instruction whose result is held in a named temporary %I0 as the *defining instruction* of %I0.
- A *label*, which indicates the location of an instruction or a basic block within the parent procedure. We use labels to reference basic blocks as well as instructions within them and we refer to the location of instructions (characterized by labels) as *program points*.
String literal (enclosed in quotation marks).

In SSA form, the scope of temporaries and labels is limited to the procedure in which they are defined. A named temporary can be thought of as a register. In ISSA form, the scope of a value, including temporaries and labels, is extended to the whole program. Certain instructions can be simplified. For instance, let us consider a temporary $%v0$, which is assigned the result of an arithmetic instruction whose operands are constant. In this case, we can evaluate the arithmetic instruction to a constant $cval$ and replace uses of $%v0$ with $cval$. We refer to the process of simplifying instructions and replacing references to their named temporary with the resulting value as folding. Moreover, we refer to values that contain memory addresses as pointer values. If a pointer value $v$ can be equal to the address of the variables $x$ and $y$, then it can be said that $v$ is aliased to $x$ and $y$.

Below, we list the instructions in our IR:

$%v0 := \text{load } v1$: is the load instruction. It assigns the contents of the memory location contained in the pointer value $v1$ to the temporary $%v0$.

$\text{store } v0,v1$: is the store instruction. It assigns the value $v1$ to the program variable whose address is contained in the pointer value $v0$.

$%v0 := \text{cpy } v1$: is the copy instruction. It assigns the value $v1$ to the temporary $%v0$.

$%v0 := \text{call } v1,v2,\ldots,vn$: is the call instruction. It calls the procedure whose address is contained in the pointer value $v1$ and passes it the arguments $v2$ up to $vn$. The return value of this instruction (if it exists) is assigned to the temporary $%v0$.

$%v0 := \text{alu } v1,v2$: is the arithmetic and logic instruction. It applies the operator $op$ on the values $v1$ and $v2$ (for operations requiring $v2$) and assigns the result to the temporary $%v0$. There are a multitude of operators that are used in this thesis. They include:
%v0 := add v1, v2: Assigns the result of v1 + v2 to %v0.

%v0 := sub v1, v2: Assigns the result of v1 − v2 to %v0.

%v0 := mul v1, v2: Assigns the result of v1 ∗ v2 to %v0.

%v0 := div v1, v2: Assigns the result of v1/v2 to %v0.

%v0 := lt v1, v2: If v1 is lower than v2 then assigns true to %v0 and otherwise assigns false to %v0.

%v0 := leq v1, v2: If v1 is lower or equal to v2 then assigns true to %v0 and otherwise assigns false to %v0.

%v0 := eq v1, v2: If v1 is equal to v2 then assigns true to %v0 and otherwise assigns false to %v0.

%v0 := neq v1, v2: If v1 is not equal to v2 then assigns true to %v0 and otherwise assigns false to %v0.

%v0 := and v1, v2: Assigns the bitwise and of v1 and v2 (v1&v2) to %v0.

%v0 := elemOf v1, v2: is the pointer arithmetic instruction. It computes the memory address of structure fields and array elements. In it, v1 is a pointer value that indicates the base (including type information) and v2 is an integer that indicates the structure field or array element.

The semantics of this instruction are that %v0 is assigned the address of the structure field or array element specified by v1 and v2. The type of the structure or array is derived by analyzing v1 and we use this information to identify the memory offset of the structure field or array element.

%v0 := select v1, v2, v3: is the selection instruction. If v1 is true, then %v0 is assigned v2. Otherwise, %v0 is assigned v3.

br v0, BB0, BB1: is the branch instruction. The instruction that will be executed next is the entry to the basic block, whose label is BB0, if v0 is true, and BB1
otherwise. If the only operand is a label $BB_0$, then this is an unconditional branch instruction. In this case, the next instruction that will be executed is the entry to the basic block whose label is $BB_0$.

**return $v_0$:** is the return instruction. It will stop executing the current procedure and resume at the program point in the code immediately after the call instruction ($ci$) which called this procedure. It sets the named temporary that is assigned $ci$ to $v_0$.

In the rest of this dissertation, we will use these instructions when presenting the IR. In addition to the instructions presented above, Section 2.5 introduces the $\phi$ instruction, which is used in SSA form and Section 3.2 introduces IR extensions that support ISSA.

In this dissertation, we reference the following sets when presenting algorithms:

- $\mathcal{VAL}$ is the set of values.
- $\mathcal{INST}$ is the set of instructions.
- $\mathcal{TMP} \subset \mathcal{VAL}$ is the set of temporaries.
- $\mathcal{PR}$ is the set of procedures.
- $\mathcal{L}$ is the set of program points.

Moreover, we assume that the following procedures are available to us:

- **InstToProgPoint** is a procedure that accepts an instruction $I \in \mathcal{INST}$ and returns its program point.

- **ProgPointToInst** is a procedure that accepts a program point $PP \in \mathcal{L}$ and returns the instruction located at $PP$.

- **InstToTemp** is a procedure that accepts an instruction $I \in \mathcal{INST}$ and returns the temporary holding the result of executing $I$. 
**TempToInst** is a procedure that accepts a temporary \( \%I0 \in T \cdot M \mathcal{P} \) and returns its defining instruction.

## 2.4 Relevant Interprocedural Analyses

In this thesis, we refer to a variety of compiler analyses and in this section we provide an overview of them. A more comprehensive description is provided by Aho et al. [2] in chapters 9 and 12.

One of the fundamental interprocedural analyses is the computation of the *call graph*, which identifies the procedures that can be called at a call instruction. The program point of a call instruction is its *call site*. A *call graph path* is a sequence of call sites \( \langle cs_1, cs_2, \ldots, cs_n \rangle \), where \( cs_{k+1} \) is within a procedure called at the call site \( cs_k \). A call graph path can be used to characterize the context under which a procedure is invoked. A procedure \( Q \) can be *reached at a call site cs*, if there exists any call graph path that starts at \( cs \) and ends at a call site targeting \( Q \). A procedure \( P \) can reach procedure \( Q \), if \( P \) contains any call site that can reach \( Q \).

A *Strongly Connected Component (SCC)* is a set of graph nodes \( S \), where each node in \( S \) can reach every node in \( S \). A call graph SCC is a set of procedures, while a *Control Flow Graph (CFG) SCC* is a set of basic blocks. An acyclic call graph is a call graph that does not contain any SCCs. We can convert a regular call graph into an acyclic call graph by identifying all the SCCs in it and collapsing each one into a single node. Hence, a node in an acyclic call graph can correspond to multiple procedures.

In this dissertation, we assume that the *reachable procedures mapping (RPC)* is already computed. In \( \mathcal{R}PC \), each call instruction \( ci \) is mapped to the set of procedures that \( ci \) can reach. Moreover, each procedure \( P \) is mapped to the set of procedures that \( P \) can reach. The mapping \( \mathcal{R}PC \) is derived using a bottom-up traversal over the acyclic call graph and reachable procedures are represented in sparse bit vectors. Moreover, we
assume the availability of the procedure $Targ$, which accepts a call instruction $ci$ and returns the set of procedures that can be called by $ci$. Sometimes programs contain indirect call instructions, which are call instructions whose pointer value is a named temporary (i.e. not a constant).

In order to identify the procedures called at indirect call instructions, we leverage a point-to graph to identify the set of procedures to which a pointer value may be equal. When a pointer value $pv$ may be equal to a given variable address $var$, we say that $pv$ points-to $var$. The point-to graph is derived by a pointer analysis [5, 22, 36, 53], that approximates the program variables to which a pointer value can be aliased. The pointer analysis can consider multiple factors in order to improve precision or provide additional information. In particular, a field-sensitive pointer analysis distinguishes between different structure fields and certain array elements whereas the field-insensitive pointer analysis does not.

### 2.5 Overview of Static Single Assignment

SSA evolved from a constant propagation algorithm [54]. Wegman and Zadeck [49] investigated a constant propagation algorithm that took conditions into account using the notion of the global value graph, which was introduced by earlier work [40]. The nodes of a global value graph represent the birthpoint of variables. The birthpoints of a given variable $v$ are its definition sites as well as confluence points, which are control flow join nodes whose predecessors have different reaching definitions for $v$. In the global value graph, an edge is constructed between each variable birthpoint and its reachable uses. During constant propagation, branches may be folded, exposing unreachable code. This can enable us to fold additional instructions to constants, since fewer variable definitions reach confluence points. Wegman and Zadeck [49] took this into account by applying the meet operator only to birthpoints that can reach a confluence point. Building on
Chapter 2. Background and Related Work

the global value graph and, in particular, the notion of birthpoints, Cytron, Lowry, and Zadeck [20] proposed a code motion algorithm that moves instructions outside of the loop.

Rather than representing the global value graph in a separate data structure, Rosen et al. [41] and Alpern et al. [4] proposed to convert the program to SSA form. In SSA form, the birthpoints for a set of program variables, which we (already stated) refer to as SSA variables are captured directly in the IR. When constructing SSA, each store instruction whose pointer value is the address of an SSA variable is converted to a copy instruction that defines a unique temporary. Each use of an SSA variable var is replaced with a unique temporary that corresponds to a definition of var. If an SSA variable is assigned more than once (or if its single assignment does not dominate all uses), then it can have confluence points at entries to basic blocks. In order to address this issue, the IR in SSA form is extended by introducing the φ instruction:

%v0 := φ (Pred1, v1), ..., (Predn, vn): is the φ instruction. It will be inserted at the entry to a basic block BB to merge the values of all predecessors of BB, which are the basic blocks (named) Pred1, ..., Predn. If the control flow graph edge from Predi to BB is taken then the value of %v0 is vi.

Dynamically, a program in SSA form may assign to the same variable multiple times. However, in the IR, each variable is assigned just once, which is why the term static single assignment was used [41]. In order to translate into SSA form, we must identify the confluence points for each variable. To address this problem, Cytron et al. [17, 18] presented an algorithm that leverages the dominance frontier to translate into SSA form efficiently.

In Figure 2.1, we illustrate the SSA form for a C program. Figure 2.1(a) presents the standard form IR, where both x and y are scalar stack variables whose address is held in the temporaries %x and %y, respectively. Since the addresses of x and y are
never taken (i.e. \( \%x \) and \( \%y \) are only used by load and store instructions, where they are the pointer value), \( x \) and \( y \) are included in the set of SSA variables. The SSA form of Figure 2.1(a) is shown in Figure 2.1(b). In it, each use of the SSA variable \( y \) is reached by a single definition (\( \text{store} \ \%y, \ldots \) in Figure 2.1(a)), whose parent is the basic block \( BB_0 \). Therefore, the temporaries \( \%v0 \) and \( \%v3 \) that use variable \( y \) are replaced with the temporary \( \%y0 \), which contains the value of the definition in the basic block \( BB_0 \). For variable \( x \), two definitions can reach the basic block \( BB_2 \): \( \text{store} \ \%x, \#20 \) if entering \( BB_2 \) from \( BB_0 \) and \( \text{store} \ \%x, \%v1 \) when entering \( BB_2 \) from \( BB_1 \). As such, \( BB_2 \) is a confluence point of variable \( x \). Because of this, we merge these two definitions by inserting a \( \phi \) instruction, whose result is assigned to the temporary \( \%x2 \). This \( \phi \) instruction is treated as a definition of the SSA variable \( x \), hence, after its execution, \( \%x2 \) contains the value of \( x \). Consequently, the temporary \( \%v2 \), which uses variable \( \%x \) and follows this \( \phi \) instruction, is replaced with the temporary \( \%x2 \) in the SSA form.

Usually copy propagation is applied during the construction of SSA or immediately
afterwards. Copy propagation replaces the temporaries that hold the value of a copy instruction with the value being copied. Moreover, each temporary holding the value of a $\phi$ instruction that merges the same value is replaced with it. For instance, in Figure 2.1(b), $%x0$ is a temporary that is assigned 20 at a copy instruction, therefore $%x0$ is replaced with 20 (at the $\phi$ instruction held in the temporary $%x2$). Moreover, $%x1$ is a temporary that is copied to $%v1$, hence $%x1$ is replaced with $%v1$.

SSA form is used in many modern compilers [26, 30]. One reason for this is that SSA construction is faster [7, 18, 43] than comparable data-flow techniques [54]. Moreover, translation out of SSA [9, 11, 18, 39, 44] is very fast as well and does not degrade code performance.

Aside from the aforementioned reasons, SSA form simplifies and improves many compiler analyses and optimizations. Wegman and Zadeck [50] build on their global value graph based constant propagation [49] by proposing an SSA-based constant propagation algorithm. Instead of querying and updating a separate data structure the revised algorithm works directly on the IR. In modern compilers, such as GCC [26] and LLVM [30], the induction variable analysis [52] is based on SSA form as well. In addition, SSA form is leveraged by a large number of compiler analyses and optimizations. This includes common subexpression elimination, partial redundancy elimination [28], code motion [41], and register allocation [10].

### 2.5.1 SSA Variables

Once a SSA form is constructed, the IR does not have any references to SSA variables as their uses are replaced with temporaries, corresponding to definitions and with values after copy propagation is performed. When constructing SSA form, compilers limit the set of SSA variables to scalar stack variables whose address is not taken. This ensures two things. First, each SSA variable $var$ is defined and used within a single procedure, because $var$ must be a scalar stack variable and its address cannot be passed to other
procedures since it is never taken. Second, since the address of var is never taken, the IR does not contain conditional accesses to var; that is, each load and store instruction either definitely accesses var or must not access var.

When extending the set of SSA variables, we increase the number of program variable uses that are replaced with a single definition. When contrasting SSA to def-use chains, extending the set of SSA variables is the equivalent of identifying def-use chains for additional program variables. As such, extending the set of SSA variables can improve optimizations as the confluence points and reaching definitions of additional program variables is available in the IR. For instance, if we extend the set of SSA variables to include var, then the constant propagation algorithm proposed by Wegman and Zadeck [50] will propagate constants through var as well. While extending the set of SSA variables can ameliorate the benefits derived from SSA-based analyses and optimizations, the construction time and memory consumption may increase.

2.6 Static Single Assignment Extensions

Many extensions have been proposed since the original work on SSA. We will review extensions that handle aliased variables, arrays, and interprocedural value propagation.

One commonly used approach to handling aliasing is to group aliased variables into sets and assign each set to a virtual SSA variable var. The uses and definitions of var will be the uses and definitions of variables within its corresponding set. Then, SSA is constructed using an algorithm such as the one proposed by Cytron et al. [18]. Note that this may change the semantics of the program. As such, virtual SSA variables as well as their uses and definitions are usually not part of the IR and are instead maintained in a data structure. When constructing SSA, the uses of virtual SSA variables will be replaced with a single definition (usually in a separate data structure and not in the IR).

In Figure 2.2, the virtual SSA variable var will represent the variables y and z. Hence,
Figure 2.2: Illustration of may def-use relations that occur when a single virtual SSA variable represents multiple program variables (y and z). In this scenario, load and store instructions whose pointer value is @var can access either y or z. When replacing uses of var with a single definition during ISSA construction, we create may def-use relations as we are not certain which program variable (either y or z) is accessed.

we replace accesses to the variables y and z in Figure 2.2(a) with the virtual SSA variable var in Figure 2.2(b). Note that in Figure 2.2(b), store instructions whose pointer value is var may assign values to either y or z, while load instructions whose pointer value is var may use either y or z. This gives rise to may def-use relations as illustrated in Figure 2.2(c).

2.6.1 Static Single Assignment Extensions That Support Aliased Variables

To accommodate store instructions to aliased variables, Cytron and Gershbein [19] introduced the IsAlias function. Conceptually, this function compares variable addresses and returns the value of an aliased SSA variable after these store instructions are executed (conditional assignment operator).

Choi et al. [15] proposed the Factored SSA (FSSA) form to save memory space when handling a large number of definitions. One issue that FSSA addresses is store instruc-
tions that may assign values to multiple variables. For each variable \textit{var} that may be assigned, a \textit{preserving definition} (instruction) is inserted to indicate that \textit{var} may be assigned a new value. Moreover, in FSSA form, a new kind of \( \phi \) instruction is introduced, which does not keep track of the values associated with incoming control flow edges. This conserves (memory) space because it does not have any operands. In order to compute the reaching definitions, an algorithm will traverse the control flow graph and retrieve the reaching definitions.

There are a few drawbacks to the above-mentioned algorithms [15, 19]. First, these algorithms do not correlate the pointer value with the accessed variables. Second, a translation out of the extended SSA form is not offered. This is important because the extended SSA form can degrade performance. For instance, inserting the \textit{IsAlias} function can have a negative impact on performance since additional branches and call instructions are executed. Moreover, the above algorithms do not describe how the side-effects of function calls are captured.

Chow et al. [16] proposed an SSA form based on global value numbering named Hashed SSA (HSSA). In this extension, a value numbering pass is first applied to number pointer values. Using alias analysis, they then determine the value numbers that alias each other and then merge each alias set into a single virtual SSA variable. The set of SSA variables will contain scalar global variables which are not aliased to other variables.

In HSSA, two additional instructions are used to identify the variables that can be assigned or used at various program points. A \( \chi \) instruction \( S \), is inserted after an instruction \( I \), that \textbf{may define} a variable \textit{var} and the operand of \( S \) is the definition of \textit{var} prior to \( I \). A \( \mu \) instruction is inserted prior to an instruction \( I \), that \textbf{may use} a variable \textit{var} and its operand is the definition of \textit{var} prior to \( I \). At call sites, \( \chi \) and \( \mu \) instructions are inserted for each regular and virtual SSA variable that can be defined or used, respectively. At each store and load instruction, whose pointer value corresponds to the virtual SSA variable \textit{var}, we insert a \( \chi \) instruction and a \( \mu \) instruction, respectively.
Afterwards, HSSA is constructed using Cytron et al. [18], by treating $\chi$ and $\mu$ instructions as store and load instructions, respectively. Moreover, HSSA collapses certain $\phi$, $\chi$, and $\mu$ instructions into “zero-version” nodes to reduce the size of the resulting IR. HSSA form is largely kept within a separate data structure, can degrade the program performance [34], and its benefit to compiler passes has not been demonstrated in the literature.

2.6.2 Array Static Single Assignment

Another relevant extension is Array SSA, which includes arrays in the set of SSA variables. Knobe and Sarkar [29] proposed an algorithm that treats each array as a single element and identifies the location where the “collapsed” array was last defined using a new IR construct. Building upon this work, Fink et al. [24] used a value numbering pass [4] to identify the heap allocated arrays and structures that are accessed at program statements. Then, similar to HSSA [16], the algorithm uses $MayDef \ (d\phi)$ and $MayUse \ (u\phi)$ instructions to represent stores and loads to arrays. Since each virtual SSA variable may correspond to multiple arrays or structures, two additional analyses are proposed. Let us assume that $I_1$ and $I_2$ are any MayDef or MayUse instructions. The definitely-same analysis is used to determine whether $I_1$ and $I_2$ must access the same variable while the definitely-different analysis determines whether $I_1$ and $I_2$ cannot access the same variable. By leveraging these analyses as well as an array subscript analysis, the proposed Array SSA form was used to remove dead code, eliminate loads and stores, and also for copy propagation. While useful for a number of analyses and optimizations, array SSA is a separate data structure instead of an IR. Moreover, the proposed Array SSA algorithms are intraprocedural and rely on the value numbering pass as well as on two analyses to derive def-use chains.
2.6.3 Prior Work on Interprocedural Static Single Assignment

Liao [33] describes an ISSA where SSA variables are alias sets (equivalence classes) computed by applying Steensgaard’s unification-based pointer analysis [46]. To generate ISSA form, Liao first represents each alias set using a single virtual SSA variable. Then, the pointer value of store and load instructions that corresponds to alias set members is replaced with the appropriate virtual SSA variable (in a separate data structure). Next, ISSA is generated using an algorithm such as the one proposed by Cytron et al. [18], where virtual SSA variables are used to propagate values across call sites. According to Staiger [45], this kind of derivation creates a greater number of merge points than if one were to use an inclusion-based pointer analysis due to its relatively lower precision. The decrease in precision has a twofold effect on construction. First, is the insertion of a greater number of spurious assignments due to a greater point-to-set size. Second, note that the call graph is derived by leveraging the pointer analysis to identify the potential indirect call instruction targets. A less precise pointer analysis would result in more edges in the call graph. Hence, more SSA variables will be propagated to redundant locations in the program.

Staiger et al. [45] used the result of the pointer analysis to map aliased variables (accessed in a given procedure) to a single virtual SSA variable. Note that a virtual SSA variable is defined and used only within a single procedure. Moreover, in each procedure, a single program variable is mapped to a different virtual SSA variable. Hence, Staiger et al. map virtual SSA variables that represent intersecting alias sets to one another at call sites. ISSA is then constructed in a separate data structure, in a similar manner to Liao [33]. Staiger et al. showed that using a more precise pointer analysis would result in dramatically less $\phi$ instructions: when using Andersen’s [5] pointer analysis rather than Steensgaard’s [46], up to 16.5 times less $\phi$ instructions were inserted.

The work by Liao [32] and Staiger et al. [45] provides a preliminary evaluation of ISSA, but it has a few drawbacks. First, ISSA is maintained in a separate data structure. This
makes it harder to leverage ISSA in compiler passes that work on SSA form. Second, neither Liao [32] nor Staiger et al. [45] perform copy propagation, which can remove false merge points and reduce the size of the IR. Furthermore, may def-use relations are present in the ISSA form and only Staiger et al. [45] mark accesses to scalar globals with must-use edges. Lastly, in contrast to our body of work, Staiger et al. [45] do not evaluate ISSA using a target application, while Liao [32] only studies the use of ISSA for an array liveness analysis.
Chapter 3

Interprocedural Static Single Assignment Form

3.1 Introduction

In Chapter 2, we reviewed the SSA form and extensions relevant to ISSA. These extensions had to address two key challenges. First, load and store instructions where the pointer value is aliased to more than one variable, including at least one SSA variable, are conditional. As a result, we cannot be certain which SSA variable is being defined or used. Second, it was necessary to propagate the values of SSA variables at call sites.

This chapter introduces our proposed ISSA, including the new instructions used to address the challenges outlined above. We use the $\phi^S$ and $\phi^L$ instructions to handle conditional load and store instructions. The $\phi^S$ instruction conditionally assigns a new value to a variable, while the $\phi^L$ instruction selectively chooses its value. Values are propagated into and out of procedures using $\phi^V$ and $\phi^C$ instructions, respectively. These new instructions are described in Section 3.2 and our proposed ISSA form is illustrated in Section 3.3 with the use of an example.

Moreover, our ISSA also enables us to identify the program-wide uses of a definition.
This is done by extending the scope of values to the whole program which requires us to define the value of named temporaries that are used outside of the procedure in which they are assigned. With this definition, we can then determine whether a $\phi^V$ or $\phi^C$ instruction that merges a single value can be folded. In Section 3.4, we present this definition, illustrate copy propagation in ISSA, and introduce terminology used in the remainder of this dissertation.

The chapter concludes with Section 3.5, which compares our ISSA form to previous work and highlights the differences.

### 3.2 IR Extensions

To construct ISSA form, we meet two challenges. First, the pointer value of load and store instructions may be equal to the address of multiple program variables. Second, we need to pass the values of SSA variables across procedures at call sites.

#### 3.2.1 Handling Aliased Program Variables

As discussed in Section 2.6, past SSA extensions took two approaches to handle aliased program variables. The first is to compare the pointer value of load and store instructions to the address of each SSA variable they may reference. In this approach, a number of comparison and branch instructions are required (as well as new basic blocks). Another approach is to create a virtual SSA variable $VirtVar$ for each group of aliased program variables $Vars$ and replace accesses to any member of $Vars$ with $VirtVar$. By doing so, we change the semantics of the program in the resulting IR. Hence, past work taking this approach maintained the original IR in order to generate a correct program.

Rather than inserting a number of new branch instructions and basic blocks for each conditional store and load instruction, we address this challenge by extending the IR with two additional instructions. We take this approach in order to reduce the size of
the IR and make conditional load and store instructions explicit. Below are the new instructions:

\[ %v0 := \phi^S pval, @var, val, curr: \]

is used to handle store instructions, where \( pval \) is the pointer value. If \( pval \) is equal to \(@var\) (the address of the SSA variable \( var \)), then \( %v0 \) is assigned \( val \). Otherwise, \( %v0 \) is assigned \( curr \).

\[ %v0 := \phi^L pval, \langle var_1, val_1 \rangle, \ldots, \langle var_n, val_n \rangle: \]

is used to handle load instructions, where \( pval \) is the pointer value. If \( pval \) is equal to \( var_i \), then the value of this instruction will be \( val_i \).

### 3.2.2 Passing Values Across Procedures

In addition to aliasing, we need to pass the values of SSA variables across procedures at call sites. We address this challenge by extending the IR with \( \phi^V \) and \( \phi^C \) instructions, which are presented below:

\[ %v0 := \phi^V \langle cs_1, val_1 \rangle, \ldots, \langle cs_n, val_n \rangle: \]

passes the value of variable \( var \) from all call instructions that target a procedure \( P \) to the entry-point of procedure \( P \). When entering procedure \( P \) from the call site \( cs_i \), the value of this instruction is \( val_i \).

\[ %v0 := \phi^C pval, \langle P_1, val_1 \rangle, \ldots, \langle P_n, val_n \rangle: \]

is inserted right after a call instruction \( ci \) and passes the value of variable \( var \) from the exit-point of all procedures called by \( ci \). The pointer value of \( ci \) is \( pval \) and if \( pval \) is equal to \( P_i \), then the value of this instruction will be \( val_i \). For direct calls, we omit the pointer value altogether.
3.3 Interprocedural Static Single Assignment Example

We present the ISSA form of the C program in Figure 3.1(a) in Figure 3.1(b). The ISSA form is derived by leveraging the \( \phi^L \), \( \phi^S \), \( \phi^V \), and \( \phi^C \) instructions presented above. In Figure 3.1(a), all four global variables \( g \), \( x \), \( y \), and \( z \) are SSA variables. As shown in Figure 3.1(c), a flow-insensitive pointer analysis indicates that \( x \) points either to \( y \) or \( z \), and that \( g \) points to \( x \). Since the dereference in Figure 3.1(a), on line 14, can access either \( y \) or \( z \), we need to insert two \( \phi^S \) instructions to handle the store, as illustrated in Figure 3.1(b) on lines 16–17. Similarly, due to the dereference on line 5 in Figure 3.1(a), we need to insert a \( \phi^L \) instruction on line 8 in Figure 3.1(b). Note that variable \( x \) is defined in procedure \( B \) and variables \( x \), \( y \), and \( z \) are used in procedure \( C \). Hence, we propagate the value of the SSA variable \( x \) out of procedure \( B \) using the \( \phi^C \) instruction whose result is assigned to \( %x_1 \) on line 15 in Figure 3.1(b). On lines 5–7 in Figure 3.1(b), we propagate the values of the global variables \( x \), \( y \), and \( z \) into procedure \( C \) by inserting the \( \phi^V \) instructions whose result is assigned to \( %x_2 \), \( %y_2 \), and \( %z_2 \), respectively.

As illustrated in Figure 3.1(d), we can fold a great number of instructions to constants by simply extending the Wegman and Zadeck [50] SSA-based constant propagation algorithm to ISSA form. First, by substituting \( %x_1 \) (line 15 in Figure 3.1(b)) with \&z we can determine that the \( \phi^S \) instructions held in the temporaries \( %y_1 \) and \( %z_1 \) are equal to 5 and 20, respectively. This allows us to replace \( %x_2 \) with \@z, \( %y_2 \) with 5, and \( %z_2 \) with 20. Then, the \( \phi^L \) instruction on line 8 in Figure 3.1(b) is replaced with the constant 20, producing the code in Figure 3.1(d).
(a) C source code.  

```
int y=5, z=10;  
int *x;  
int **g;  
static void C( ) {  
    print( **g );  
}  
static void B( ) {  
    *g = &z;  
}  
void main( ) {  
g = &x;  
x = &y;  
CI1: B( );  
**g = 20;  
CI2: C( );
}
```

(b) ISSA form for Figure 3.1(a) after \( \phi^S \), \( \phi^L \), \( \phi^V \), and \( \phi^C \) instructions are inserted.  

```
int y, z;  
int* x;  
int** g;  
static void C( ) {  
    %x2 := \( \phi^V \) \langle CI2,%x1 \rangle ;  
    %y2 := \( \phi^V \) \langle CI2,%y1 \rangle ;  
    %z2 := \( \phi^V \) \langle CI2,%z1 \rangle ;  
    %v0 := \( \phi^L \) %x2 , \langle @y,%y2 \rangle , \langle @z,%z2 \rangle ;  
    call @print, %v0;  
}  
static void B( ) { }  
void main( ) {  
    CI1: call @B;  
    %x1 := \( \phi^C \) \langle @B,%z \rangle ;  
    %y1 := \( \phi^S \) x1,@y,#20,#5 ;  
    %z1 := \( \phi^S \) x1,@z,#20,#10;17  
    CI2: call @C;  
}
```

(c) Point-to graph for Figure 3.1(a).  

(d) ISSA form after copy propagation and constant folding are applied.  

```
static void C( ) {  
call @print, #20;  
}  
void main( ) {  
    call @C;  
}
```

Figure 3.1: Interprocedural SSA example.
3.4 Copy Propagated ISSA

Within our framework, the scope of temporaries is program wide, thus enabling us to fold $\phi^V$ and $\phi^C$ instructions. The benefits of this approach are IR size reduction and the removal of false merge points.

In an SSA form program, a named temporary cannot be used outside of the procedure in which it is defined. Let us consider instruction $I$ in procedure $P$ whose result is assigned to a temporary $%I0$. In this scenario, at a given usage program point $U$ in the $n$th invocation of $P$, $%I0$ is equal to the result computed by the last instance of $I$ (when reaching program point $U$) in the $n$th invocation of $P$. When procedure $P$ is part of a SCC, then the last instance of $I$ at $U$ may be in invocation $m$ of $P$, where $m > n$. Because of this, values of temporaries are typically saved on the stack when executing call instructions and are restored when returning from them. In this example, note that $%I0$ would have to be a global variable in order for it to hold the result of the last instance of instruction $I$.

In order to use the temporary $%I0$ outside of $P$, we must be able to identify the instance of $I$ that $%I0$ is equal to at a usage point. To this end, we provide the following definition:

$%I0$ holds the value computed by the instance of $I$ in the most recent invocation of $P$ that did not yet return (still has a corresponding call frame on the stack). Alternatively, if all invocations of $P$ returned ($P$ does not have a call frame on the stack), then $%I0$ holds the value computed by the last instance of $I$ in the last invocation of $P$ that returned.

Based on this definition, the value of $%I0$ varies with its usage points, but at any program point in $P$, it is identical in both SSA and ISSA. As such, each program in SSA form is a valid ISSA form program. This property simplifies the construction of ISSA form, since we do not have to revise the (SSA form) IR prior to constructing ISSA.
3.4.1 Folding $\phi^C$ Instructions

It may seem intuitive that $\phi^V$ and $\phi^C$ instructions, which merge a single value, can always be folded. However, when a $\phi^C$ instruction in procedure $P$ merges a temporary defined in procedure $Q$, this is not always possible. In this section, we illustrate that folding such a $\phi^C$ instruction also depends on its usage points as well as whether $Q$ can reach $P$ (i.e., $P$ and $Q$ are part of a call graph SCC).

Let us consider a $\phi^C$ instruction $I$, whose result is assigned to the temporary $I0$. Let us further assume that $I$ merges a single temporary $J0$, which is defined in procedure $P_j$. Replacing $I0$ with $J0$ is not always legal, as the invocation of $P_j$ to which $J0$ corresponds depends on the usage point of $I0$. This is illustrated in Figure 3.2. In Figure 3.2(a), the first and second return values from procedure $Sum$ are printed on line 8. In the ISSA form of Figure 3.2(a), which is shown in Figure 3.2(b), these values are propagated into procedure $main$ using the $\phi^C$ instructions defining $v4$ and $v5$. Note that $v5$ is equal to $v3$ on line 12 in Figure 3.2(b), because procedure $Sum$ cannot...
be reached by any other call instruction between the definition of \( v_5 \) and the \textit{print} call on line 12. We can optimize the program by replacing \( v_3 \) with \( v_5 \), eliminating both a temporary and its defining (\( \phi^C \)) instruction. Moreover, note that \( v_4 \) is not equal to \( v_3 \) on line 12 because procedure \textit{Sum} is called in between at \( CI2 \) (on line 10).

In Figure 3.3, we provide the C source code and partial ISSA form for a recursive procedure that computes Fibonacci numbers. In this example, the temporary \( vout \) on line 19 in Figure 3.3(b), is passed via the two \( \phi^C \) instructions defining \( v_7 \) and \( v_9 \) on lines 11 and 14, respectively. Note that \( v_7 \) and \( v_9 \) are assigned a temporary (\( vout \)) which is defined in invocations of \textit{fibonacci} that have already returned. Moreover, \( v_7 \) and \( v_9 \) are temporaries that are defined within the procedure \textit{fibonacci}. Hence, at each usage point of both \( v_7 \) and \( v_9 \), the invocation of \textit{fibonacci} in which \( v_7, v_9, \) and \( vout \) are defined will be the same. As such, even though the \( \phi^C \) instructions defining the temporaries \( v_7 \) and \( v_9 \) merge the single value \( vout \), neither \( v_7 \) nor \( v_9 \) are equal to \( vout \) at any program point. Therefore, neither \( v_7 \) nor \( v_9 \) can be replaced with \( vout \).

### 3.4.2 Example and Terminology

In this section, we provide an example to illustrate copy propagated ISSA and introduce new terminology. In the source code shown in Figure 3.4(a), the dereference of the variable \( ptr \) is incremented with the difference between the return value of the first and second calls to procedure \textit{getPercentage}. The SSA form for the C source code in Figure 3.4(a) is presented in Figure 3.4(b). Note that the use of \( ptr \) on line 19 in Figure 3.4(a) is replaced with the \( \phi \) instruction whose result is assigned to \( v_5 \) in Figure 3.4(b), line 17. Hence, the dereference of \( ptr \) can access either \( y \) or \( z \).

The ISSA form for Figure 3.4 is given in Figure 3.5. In Figure 3.5(a), we illustrate the ISSA form prior to applying copy propagation. In order to propagate the parameters \( x \) and \( total \) into procedure \textit{getPercentage}, the \( \phi^V \) instructions whose result is assigned
int fibonacci(int Num) {  
    int Result;  
    if (Num <= 2) {  
        Result = 1;  
    } else {  
        int Res1 = fibonacci(Num-1);  
        int Res2 = fibonacci(Num-2);  
        Result = Res1 + Res2;  
    }  
    return Result;  
}  

int main() {  
    int Res5 = fibonacci(5);  
    print(Res5);  
}
Figure 3.4: C source code and its associated SSA form. Note that $%v5$ in Figure 3.4, line 17, is either equal to @z or @y, hence its dereferences (on line 20 and line 23) can access the variables y and z.
There are two $\phi^C$ instructions in this example. The temporary $\%v7$ is assigned the result of a $\phi^C$ instruction that merges the temporary $\%v16$, which is defined in procedure getPercentage. We can replace $\%v7$ with $\%v16$ because three conditions are satisfied. First, the defining instruction of $\%v7$ merges a single value ($\%v16$). Second, procedures main and getPercentage are not in the same SCC. Thus, at every use in procedure main, $\%v7$ corresponds to a call frame of getPercentage that was popped of the stack rather than a call frame of getPercentage on the stack. Third, procedure getPercentage cannot be reached on any path between the program point where $\%v7$ is defined and its use on line 27 in Figure 3.5(b). Therefore, $\%v16$ would be equal to $\%v7$ on line 27.

Note that the $\phi^C$ instruction whose value is held in $\%v6$ satisfies the first two conditions outlined above as well. However, the third condition is not satisfied since procedure getPercentage can be reached at the call site $CI2$ in Figure 3.5(b), line 23. The call site $CI2$ is located on a path between the definition of $\%v6$ on line 22 in Figure 3.5(b) and its use in the addition instruction, whose result is assigned to $\%v9$ on line 26. Therefore, $\%v16$ would not be equal to $\%v6$ on line 26 under our definition ($\%v16$ would be equal to $\%v7$ instead).

Using Figure 3.5, we illustrate a number of key terms that are defined below:

**Interprocedural reference:** A reference from an instruction located in a procedure $P$ to one of its operands, a named temporary that is defined in another procedure $Q \neq P$. In Figure 3.5(b), the reference from the defining instruction of $\%v10$ (located on line 27 in procedure main) to its operand, the temporary $\%v16$ (located on line 7 in procedure getPercentage) is an interprocedural reference. Moreover, the defining instruction of $\%v16$ has an interprocedural reference to $\%v3$.

**Propagation point:** The call site or procedure entry through which a temporary is propagated into the parent procedure of an instruction using it. For example, the propagation point of the interprocedural reference from the defining instruction of
Chapter 3. Interprocedural Static Single Assignment Form

38

int getPercentage(int %x, int %total) {
    %v13 := φV (CI1,%v1), (CI2,%v2);  
    %v14 := φV (CI1,%v3), (CI2,%v3);  
    %v15 := mul %v13,#100  
    %v16 := div %v15,%v14;  
    return %v16;  
}

void main() {

    %v1 := call @getI;  
    %v2 := call @getI;  
    %v3 := call @getI;  
    %v4 := lt %v1,%v3;  
    br %v4, BB1,BB2;  
    BB1: call @getPercentage, %v1,%v3;  
    %v6 := φC @getPercentage,%v16;  
    %v7 := φC @getPercentage,%v16;  
    %v8 := φL %v5, @y,#10, @z,#20;  
    %v9 := add %v8,%v6;  
    %v10 := sub %v9,%v7;  
    %v11 := φS %v5, @y, %v10, #10;  
    %v12 := φS %v5, @z, %v10, #20;  
    return %v16;  
}

(a) ISSA form derived from the SSA form in Figure 3.4(b).

(b) ISSA form after copy propagation is applied.

Figure 3.5: ISSA form for the code shown in Figure 3.4.
Chapter 3. Interprocedural Static Single Assignment Form

%v10 to %v16 on line 27 of Figure 3.5(b) is the call site CI2. In another example, the propagation point of the interprocedural reference from the defining instruction of %v16 to %v3 on line 7 of Figure 3.5(b) is the entry to procedure getPercentage.

3.5 Comparison

Various approaches have been proposed to handle aliasing and value passing across call sites. The $\phi^S$ instruction we have proposed is very similar to the IsAlias function described by Cytron and Gershbein [19]. However, in their SSA form, dereference-based assignments that can assign values to SSA variables are kept in the IR and will be executed at runtime. Furthermore, both within the IsAlias function and the IR, SSA variables are loaded through dereferencing pointers. In contrast, our ISSA form makes conditional assignments and loads explicit; using $\phi^S$ and $\phi^L$ instructions we can immediately identify the pointer value, the variables it is aliased to, and their values.

Chow et al. [16], Liao [32], and Staiger et al. [45] do not keep track of the pointers when assigning or loading values to aliased variables. In the ISSA form proposed by Liao [32] and Staiger et al. [45], we can identify the call sites associated with a given incoming or outgoing value, but copy propagation is not applied. Moreover, in the work of Liao [32] and Staiger et al. [45] may def-use relations arise (see Section 2.6 for details), since a single virtual SSA variable represents multiple program variables.

The presence of may def-use relations forces us to update compiler passes and thus complicates out-of-ISSA translation. This is illustrated using the example shown in Section 2.6. In Figure 2.2, var replaces accesses to the variables $y$ and $z$. As a result, we cannot determine whether we are assigning or referencing variables $y$ or $z$, thus making it impossible to revert back the program in Figure 2.2(b) to the original IR in Figure 2.2(a).

For these reasons, ISSA is an auxiliary representation of the program in previous work. However, non-IR ISSA has a number of drawbacks. First, we have to maintain
and update a mapping between instructions and the data structure representing ISSA. The maintenance for such a mapping consumes memory and forces us to update certain compiler passes. In fact, to leverage ISSA, compiler passes need to reference both the IR and the data structure that is representing ISSA. Hence, SSA-based compiler passes need to be modified further. During its development, SSA form [4, 18, 41] has evolved from the global value graph, which is a data structure that represents birthpoints [20, 40, 49]. When analyzing the tradeoffs, modern compilers adapted SSA as an IR.

Ultimately, there are four significant differences between our ISSA and the ISSA found in the literature. First, by keeping track of the pointer value, we can fold $\phi^S$ and $\phi^L$ instructions. Second, we remove false merge points and save memory by using copy propagation to fold $\phi^V$ and $\phi^C$ instructions. This extends the scope of values to the whole program and, as such, interprocedural def-use chains are explicit in our proposed ISSA. Third, the ISSA we use does not contain may def-use relations, hence, less effort is required to leverage it in compiler passes. Finally, our ISSA is directly available to compiler passes because it is an IR rather than a separate data structure.
Chapter 4

Interprocedural Static Single Assignment Construction

4.1 Introduction

In this chapter, we describe how the proposed ISSA form is constructed from SSA form. A high-level flow diagram illustrating this process can be found in Figure 4.1. The process is also explained below.

The point-to function is necessary to identify the set of SSA variables that may be accessed through pointer dereferences. Formally, we use the function $PT$, which maps a pointer value to the set of program variables it may point-to. The point-to function is derived using a field-sensitive pointer analysis described in Section 4.2.

In addition to the point-to function, we also need to identify the subset of program variables for SSA conversion, $Vars$, called the SSA variables. These include structure fields and scalars in: global variables, stack allocated variables in non-recursive procedures, and singleton heap variables, which are allocated by call instructions that are executed once at most. We identify singleton heap variables using the invocation count analysis that computes $AllocatedOnce$, which is the set of heap allocation instructions
that are executed once at most. In Section 4.3, we detail the additional SSA variables and describe the invocation count analysis.

After the set of SSA variables is chosen, we visit load and store instructions and use the point-to function $PT$ to identify the SSA variables that are accessed at these instructions. When a store or a load instruction accesses a single variable $var$, then we replace its pointer value with $var$. Otherwise, we insert $\phi^S$ and $\phi^L$ instructions. The dereference conversion is described in more detail in Section 4.4.

Once all dereferences are converted, the program will not contain any load or store instructions that conditionally access SSA variables. That is, the pointer value of load and store instructions is either equal to the address of an SSA variable or is not aliased to any SSA variable. Hence, we compute the set of SSA variables accessed in each procedure by using just an IR traversal. A flow-insensitive bottom-up traversal over the call graph will determine the set of SSA variables that are defined ($\text{MOD}$) and used ($\text{REF}$) in each procedure. These sets are then used to insert $\phi^V$ and $\phi^C$ instructions which propagate
the values of variables in and out of procedures. In order to reduce the number of $\phi^V$ and $\phi^C$ instructions that are inserted, we prune $\text{MOD}$ and $\text{REF}$ by leveraging an ISSA liveness analysis that identifies variables whose value does not have to be propagated into procedures or out of them. We present our algorithm for placing $\phi^V$ and $\phi^C$ instructions in Section 4.5.

Following these steps, we allocate and place $\phi$ instructions. We treat the newly inserted $\phi^S$, $\phi^V$, and $\phi^C$ instructions as stores and the $\phi^L$ instructions as loads. By applying the algorithm proposed by Cytron et al. [18], we place $\phi$ instructions at the confluence points of the new SSA variables. In the last step, we perform copy propagation by folding $\phi^V$ and $\phi^C$ instructions. The algorithm for doing this is presented in Section 4.6.

In various compiler passes, we may need to replace a given temporary $\%I0$ (that holds the value computed by an instruction) with another temporary $\%J0$. In Section 4.7, we present a data structure that is leveraged to check whether it is legal to perform this replacement at each use of $\%I0$. In Section 4.8, we evaluate the construction of ISSA form experimentally, on a set of MediaBench [31] and SPECINT2000 [1] benchmarks.

Compared to previous ISSA construction algorithms [32, 45], we use a more precise pointer analysis and employ techniques to extend the set of SSA variables, reduce the insertion of redundant instructions, and remove false merge points. More specifically, we make the following contributions:

- We quantify why the previous approach, in which a field-insensitive pointer analysis is used and only strong updates to scalar globals are handled (similar to Staiger [45]), is less effective. By handling structure fields and certain heap locations, we replace on average 2.2 times more load instructions with the definition of the SSA variables they reference. In addition, we demonstrate that the field-sensitive pointer analysis reduces the number of SSA variables propagated in and out of procedures by a factor of 12.2, on average, when compared to the field-insensitive pointer analysis.
• We propose a copy propagation algorithm that removes 44.5% of the $\phi^V$ and $\phi^C$ instructions that are inserted.

• We propose an ISSA liveness analysis and leverage it to reduce the SSA variables propagated in and out of procedures. By using this technique, we reduce the number of $\phi^V$ and $\phi^C$ instructions that would have been inserted by 24.8%.

This chapter concludes with a summary in Section 4.9.

4.2 Pointer Analysis

Our pointer analysis is inclusion-based and field-sensitive. It does not take the procedure context or execution path into account (i.e. it is context-insensitive and flow-insensitive). We process the SSA IR and generate constraints as well as the initial point-to graph, using the Yong et al. [53] algorithm. For each heap allocation site, we create a different object; this enables us to distinguish between heap variables that are allocated at different allocation sites. Some heap variables are allocated by calling memory manager procedures. We treat call instructions that target these procedures as allocation instructions. This enables us to distinguish between heap variables that are allocated using calls to the memory manager at different sites.

We first collapse cycles [23] and then proceed to solve the constraints incrementally. We distinguish between each field in a structure and each element in small arrays (less than 20 elements). Pearce [36] has a similar pointer analysis which is available in GCC [26]. The call graph is built iteratively, by using the intermediate point-to graph (computed after each iteration) to identify the procedures called at indirect call instructions. When a new call edge is discovered, we add constraints to the pointer analysis which copy the pointer value of arguments (from the call instruction) to the parameters of newly targeted procedure.
Chapter 4. Interprocedural Static Single Assignment Construction

4.3 Choosing SSA Variables

Recall that for intraprocedural SSA, the set of SSA variables consists of scalar stack variables whose address is not taken. In ISSA, the set of SSA variables also includes the following program variables:

- Global variables.

- Stack variables in non-recursive procedures. We use the call graph to identify the set of recursive procedures and exclude stack variables within them.

- Heap variables which are allocated by call instructions that are executed once at most. We refer to these variables as singleton heap variables.

- Scalars and structure fields for all variable types described above (i.e. globals, stack variables, and singleton heap variables). Only scalars and structures (structure within structure) fields are included in the set of SSA variables within each structure (i.e. we do not include any arrays).

In order to identify singleton heap variables, we first compute the set of procedures executed more than once \((MultipleInvoked)\) as described in Section 4.3.1. Then, using the set \(MultipleInvoked\) we derive \(AllocatedOnce\) as outlined in Section 4.3.2.

4.3.1 Invocation Count Analysis

The set of procedures that can be invoked multiple times \((MultipleInvoked)\) is identified by using Algorithm 4.1. The input to Algorithm 4.1 is the program as well as \(ProcsInSCC\) and \(BBsInSCC\) which are the set of procedures in call graph SCCs and the set of basic blocks in control flow graph SCCs, respectively. Moreover, Algorithm 4.1 also receives the mapping \(RPC\) as input, which was defined in Section 2.4, and can be used to identify the set of procedures a call instruction or procedure can reach. At first,
Algorithm 4.1 Computes the set of procedures that may be invoked more than once ($\text{MultipleInvoked}$).

**Input:** $\text{ProcsInSCC}, \text{BBsInSCC}, \mathcal{RPC}$  
**Output:** $\text{MultipleInvoked}$

1: $\text{MultipleInvoked} := \text{ProcsInSCC}$  
2: foreach procedure $Q \in \text{ProcsInSCC}$ do  
3: $\text{MultipleInvoked} := \text{MultipleInvoked} \cup \mathcal{RPC}[Q]$  
4: foreach basic block $BB \in \text{BBsInSCC}$ do  
5: foreach call instruction $ci \in BB$ do  
6: $\text{MultipleInvoked} := \text{MultipleInvoked} \cup \mathcal{RPC}[ci]$  
7: foreach procedure $P \notin \text{MultipleInvoked}$ do  
8: $\text{ReachProcsSum} := \emptyset$  
9: foreach node $N$ in a topological traversal over the acyclic CFG of $P$ do  
10: $\text{ReachProcsSum}[N] := \bigcup_{M, \text{predecessor of } N} \text{ReachProcsSum}[M]$  
11: if $N$ is not a SCC then  
12: Let us assume $BB$ is the single basic block in $N$  
13: foreach call instruction $ci \in BB$ do  
14: $\text{Reached} := \mathcal{RPC}[ci]$  
15: $\text{MultipleInvoked} := \text{MultipleInvoked} \cup (\text{Reached} \cap \text{ReachProcsSum}[N])$  
16: $\text{ReachProcsSum}[N] := \text{ReachProcsSum}[N] \cup \text{Reached}$

Algorithm 4.1 adds all of the procedures in call graph SCCs and the procedures called from control flow graph SCCs to $\text{MultipleInvoked}$. Furthermore, note that each procedure $P$ that is reached from a procedure in $\text{MultipleInvoked}$ may also be called multiple times; hence we add $P$ to $\text{MultipleInvoked}$ as well. Afterwards, we apply a topological traversal over the acyclic control flow graph of each procedure $P \notin \text{MultipleInvoked}$, to identify procedures that are called more than once on a given path. For each basic block $BB$, we first identify the set of procedures reachable from call instructions executed on a path to it on line 10. In order to derive this set, we maintain the set of procedures reached after each SCC component, in the mapping $\text{ReachProcsSum}$. We identify the set of procedures reached on any path to $BB$ by taking the union of procedures reached on paths to predecessors of $BB$. Once this step is performed, each call instruction $ci$ in $BB$ is visited and we identify $\text{Reached} = \mathcal{RPC}[ci]$. Procedures in $\text{Reached} \cap \text{ReachProcsSum}[N]$ (where $N$ is the SCC component to which $BB$ belongs) can be reached more than once on a given path and as such, we add these procedures to $\text{MultipleInvoked}$. Finally, we
add $\text{Reached}$ to $\text{ReachProcsSum}[N]$ to keep track of the reachable procedures. Note that if another call instruction in $BB$ can reach a procedure $Q \in \text{Reached}$, then $Q$ will be added to $\text{MultipleInvoked}$.

### 4.3.2 Heap Allocation Sites Executed Once At Most

**Algorithm 4.2** Compute $\text{AllocatedOnce}$, which is the set of heap allocation instructions that are executed once at most.

**Input:** $\text{MultipleInvoked}$, $\text{BBsInSCC}$

**Output:** $\text{AllocatedOnce}$

1: $\text{AllocatedOnce} := \emptyset$
2: foreach procedure $P \notin \text{MultipleInvoked}$
3:   foreach basic block $BB \in P$ where $BB \notin \text{BBsInSCC}$
4:     foreach instruction $I \in BB$ do
5:       if $I$ is a heap allocation instruction then
6:         $\text{AllocatedOnce} := \text{AllocatedOnce} \cup I$

Algorithm 4.2 is used to derive $\text{AllocatedOnce}$, which is the set of heap allocation instructions that are executed once at most. It accepts as input the set of procedures executed multiple times ($\text{MultipleInvoked}$) and the set of basic blocks in SCCs ($\text{BBsInSCC}$). Then, all heap allocation instructions whose parent is not in $\text{BBsInSCC}$ and whose parent procedure is not in $\text{MultipleInvoked}$ are added to $\text{AllocatedOnce}$.

### 4.4 Dereference Conversion

Recall that $\text{Vars}$ is the set of SSA variables, $\text{PT}$ is the point-to function, and let us assume $I$ is a load or store instruction whose pointer value is $pv \notin \text{Vars}$ and $\text{PT}(pv) \cap \text{Vars} \neq \emptyset$. Dereference conversion will either replace $pv$ with the SSA variable it points-to, insert a sequence of $\phi^S$ instructions, or insert a $\phi^L$ instruction.

If $I$ is a load instruction $\%I_0 := \text{load } pv$, then we apply Algorithm 4.3 to convert the dereference. First, we check whether $pv$ points-to a single SSA variable $pvar$ on line 1 and replace $pv$ with $pvar$, if this is the case. Otherwise, we replace $\%I_0$ with $\%J_0$,
Algorithm 4.3 Dereference conversion for a load instruction.

Input: $\mathcal{PT}, Vars$, the load instruction $I$: $%I0 := load pv$

Require: $\mathcal{PT}(pv) \cap Vars \neq \emptyset$

1: if $|\mathcal{PT}(pv)| = 1$ then
2: set the pointer value of $I$ to $pvar \in \mathcal{PT}(pv)$
3: else
4: insert a new instruction $J$: $%J0 := \phi^L pv$
5: foreach $var \in \mathcal{PT}(pv) \cap Vars$ do
6: insert a new instruction: $%varL := load var$
7: add $\langle var, %varL \rangle$ to $J$
8: if $\mathcal{PT}(pv) \setminus Vars \neq \emptyset$ then
9: insert a new instruction: $%defL := load pv$
10: add $\langle Default, %defL \rangle$ to $J$
11: replace $%I0$ with $%J0$

the temporary holding the value computed by the $\phi^L$ instruction $J$ created on line 4.

The operands of $J$ are the addresses and values of the variables in $\mathcal{PT}(pv) \cap Vars$. If $\mathcal{PT}(pv) \setminus Vars \neq \emptyset$, then $\mathcal{PT}(pv)$ contains non-SSA variables.

Note that only uses of SSA variables are replaced with a temporary during ISSA construction. We do not identify the reaching definitions of non-SSA variables nor insert $\phi$ instructions for them. Hence, if $\mathcal{PT}(pv) \setminus Vars \neq \emptyset$ we insert a load instruction whose pointer value is $pv$ right before $I$ and assign its result to the temporary $%defL$. Then $%defL$ is added to the $\phi^L$ instruction $J$ as the default value; when $pv$ is not equal to the address of any SSA variables (i.e. $\mathcal{PT}(pv) \cap Vars$) then $%J0$ is assigned $%defL$.

Algorithm 4.4 Dereference conversion for a store instruction.

Input: $\mathcal{PT}, Vars$, the store instruction $I$: store $pv, val$

Require: $\mathcal{PT}(pv) \cap Vars \neq \emptyset$

1: if $|\mathcal{PT}(pv)| = 1$ then
2: set the pointer value of $I$ to $pvar \in \mathcal{PT}(pv)$
3: else
4: foreach $var \in \mathcal{PT}(pv) \cap Vars$ do
5: insert a new instruction: $%varL := load var$
6: insert a new instruction: $%J0 := \phi^S pv, var, val, %varL$

If $I$ is a store instruction, we apply Algorithm 4.4 to convert the dereference. Similar to Algorithm 4.3, if $pv$ points-to a single SSA variable, then we replace $pv$ with it. Otherwise,
we insert a series of $\phi^S$ instructions. For each SSA variable $\text{var} \in PT(pv) \cap \text{Vars}$, with a current value $\text{curr}$, we insert the instruction $\phi^S_{pv,\text{var},\text{val},\text{curr}}$.

To model the impact (on SSA variables) of a call instruction $ci$ that invokes a library procedure $P$, we insert store, load, $\phi^S$, and $\phi^L$ instructions. In cases where the impact of a library procedure $P$ cannot be accurately predicted, we identify the set of SSA variables that may be used or defined by the library procedure invocation. Then, we write the value of SSA variables that may be used by inserting store instructions prior to $ci$. Moreover, to retrieve the value of each SSA variable $\text{var}$ (whose address is $@\text{var}$) that may be assigned within $P$ we insert a load instruction $LI$ with a pointer value $@\text{var}$, right after $ci$. When constructing ISSA, the load instruction $LI$ is treated as a definition of $\text{var}$. All store and load instructions that are inserted due to library calls are marked using flags and are removed during the out-of-ISSA translation, presented in Chapter 5.

Once ISSA form is constructed, copy propagation can expose a number of pointer values that can be simplified. This can be leveraged to refine the results of the pointer analysis since we can fold $\phi^L$ and $\phi^S$ instructions. Moreover, we can capture the impact of program transformations, such as cloning and inlining on pointer values.

**Example 4.1 Converting Dereferences in Figure 3.1**

Note that in Figure 3.1(b), on lines 16–17 we insert two $\phi^S$ instructions that conditionally assign the value 20 to the SSA variables $y$ and $z$. They are inserted because, according to the point-to function, the store instruction in Figure 3.1(a) on line 14 can store 20 to either SSA variable.

In Figure 3.1(b), line 8, we insert a $\phi^L$ instruction. It is inserted because, according to the point-to function, the load instruction in Figure 3.1(a) on line 5 can access either SSA variable $y$ or $z$.

On lines 5 and 14 in Figure 3.1(a), $g$ is dereferenced twice. Since $g$ points-to $x$, we replace the pointer value of $*g$ with the address of $x$. Hence, in Figure 3.1(b), $*g$ is replaced with a load of variable $x$ on line 8 and lines 16–17.
4.5 Inserting $\phi^V$ and $\phi^C$ Instructions

The focus of this section is the insertion of $\phi^V$ and $\phi^C$ instructions. This is done by first computing the set of SSA variables referenced and modified in each procedure. Next, we place $\phi^V$ and $\phi^C$ instructions to propagate the values of SSA variables across call sites. Moreover, we avoid inserting $\phi^V$ and $\phi^C$ instructions that propagate the values of redundant SSA variables. For each procedure $P$, this is done by computing the set of SSA variables that may be defined prior to entering $P$ and may be used after exiting $P$.

4.5.1 Procedure Mod/Ref Analysis

In the mappings $\text{REF}$ and $\text{MOD}$, we map each procedure $P$ to the set of SSA variables that may be used or defined in $P$, respectively. In order to derive these sets, Algorithm 4.5 applies a postorder (bottom-up) traversal over the acyclic call graph. Recall that each acyclic call graph node can correspond to multiple procedures. When visiting a node $N$ that contains a procedure $P$, we update the mappings $\text{MOD}[P]$ and $\text{REF}[P]$ with the set of SSA variables defined and used by procedures reachable from $P$.

The intraprocedural pass uses a flow-insensitive algorithm to compute the set of SSA variables that are defined and used within each procedure. This result is refined using the ISSA liveness analysis presented in Section 4.5.2. In the intraprocedural pass on lines 3–16 in Algorithm 4.5, we iterate over each instruction $I$ in each procedure $P \in N$ and update $\text{LREF}$ ($\text{Local REF}$) and $\text{LMOD}$ ($\text{Local MOD}$) with the SSA variables that are used and defined when executing $I$. When $I$ is a load or $\phi^L$ instruction we update $\text{LREF}$ and if $I$ is a store or $\phi^S$ instruction, we update $\text{LMOD}$. If $I$ is a call instruction, we update $\text{LREF}$ and $\text{LMOD}$ with the set of SSA variables used and defined in each procedure reached from $I$, respectively. This is accomplished by performing queries on $\text{REF}$ and $\text{MOD}$ entries of each procedure $Q \notin N$ reached from $I$.

Note that the mappings $\text{REF}$ and $\text{MOD}$ for procedure $Q$ were already derived,
Algorithm 4.5 Procedure Mod/Ref Analysis.

Input: Acyclic Call Graph (ACG), RPC
Output: MOD and REF

1: foreach node N in a postorder traversal over ACG do
2:  \text{LREF} := \text{LMOD} := \emptyset
3: foreach procedure P \in N do
4:  foreach instruction I in procedure P do
5:    if I = load \text{var} \land \text{var} \in Vars then
6:      \text{LREF} := \text{LREF} \cup \text{var}
7:    else if I = store \text{var}, \text{val} \land \text{var} \in Vars then
8:      \text{LMOD} := \text{LMOD} \cup \text{var}
9:    else if I = \phi^{L} pv, \langle \text{var}_{1}, \text{val}_{1} \rangle, \ldots, \langle \text{var}_{n}, \text{val}_{n} \rangle then
10:      foreach \text{var}_{i} \in Vars, 1 \leq i \leq n do
11:        \text{LREF} := \text{LREF} \cup \text{var}_{i}
12:    else if I = \phi^{S} pv, \text{var}, \text{val}, \text{curr} then
13:      \text{LMOD} := \text{LMOD} \cup \text{var}
14:    else if I = call pv, \ldots then
15:      foreach procedure Q \in RPC[I] \land Q \notin N do
16:        \text{LREF} := \text{LREF} \cup \text{REF}[Q], \text{LMOD} := \text{LMOD} \cup \text{MOD}[Q]
17:      foreach procedure P \in N do
18:        \text{REF}[P] := \text{LREF}, \text{MOD}[P] := \text{LMOD}

because we are applying a postorder traversal over an acyclic call graph. If N contains multiple nodes, then each procedure in P \in N can reach any other procedure Q \in N – P. Hence, Algorithm 4.5 adds up all SSA variables that are used and defined within procedures in N in the sets LREF and LMOD, respectively. Then, for each procedure P \in N, LREF and LMOD are assigned to \text{REF}[P] and \text{MOD}[P], respectively.

Example 4.2 Computing \text{REF} and \text{MOD} for the examples in Figure 3.1 and Figure 3.5

In Figure 3.1, procedure C will have load instructions for the SSA variables x, y, and z due to the insertion of the \phi^{L} instruction on line 8 in Figure 3.1(c). Hence, \text{REF}[C] = \{x, y, z\}. Because no \phi^{S} or store instructions are present in procedure C, \text{MOD}[C] = \emptyset. Procedure B consists of only a store instruction to SSA variable x. Thus, \text{REF}[B] = \emptyset and \text{MOD}[B] = \{x\}.

Let us now focus on the example in Figure 3.5. Since procedure getPercentage does
not use or define any SSA variable, both \( \mathcal{R}_P[\text{getPercentage}] = \varnothing \) and \( \mathcal{M}_P[\text{getPercentage}] = \varnothing \).

### 4.5.2 ISSA Liveness Analysis

Algorithm 4.6 removes SSA variables that do not have to be propagated in and out of a given procedure \( P \) from \( \mathcal{R}_P[P] \) and \( \mathcal{M}_P[P] \), respectively. To accomplish this, we make two observations. First, SSA variables that are not used after exiting procedure \( P \) do not have to be propagated out of \( P \). Second, each SSA variable \( \text{var} \) that is not defined prior to entering \( P \) does not have to be propagated into \( P \). By not inserting a \( \phi^V \) instruction for \( \text{var} \) at the entry to \( P \), \( \text{var} \) will be associated with an undefined value at the entry to \( P \). Note that this preserves the semantics of the program while reducing the number of \( \phi^V \) instructions that are inserted during ISSA construction.

While liveness analysis focuses on the uses of variables rather than their definitions, identifying undefined SSA variables enables us to reduce the number of \( \phi^V \) instructions without resorting to more computationally expensive algorithms. One possible liveness analysis algorithm is the extension of the intraprocedural liveness analysis outlined by Aho et al. [2] to the whole program. In such an extension, we would have to maintain the set of live variables for each basic block in every procedure and update the live sets as we iterate multiple times over the whole program until a fixed point is reached. The algorithm we propose requires less memory (we maintain just two sets per procedure) and just one iteration while handling a number of important scenarios. One important scenario involves global variables which are defined and used within a small set of procedures (usually within a single file). In such a scenario, \( \phi^V \) instructions typically have to be inserted just within these procedures.

In Figure 4.2, we provide an example that illustrates a scenario where a flow-sensitive interprocedural liveness analysis improves precision over the proposed ISSA liveness analysis. In the program shown in Figure 4.2, the proposed algorithm will conclude that the
global variables \( a \), \( b \), and \( c \) have to be propagated into procedure \( \text{proc} \) because they are defined prior to some invocation of \( \text{proc} \) (by the call to \( \text{initGlobals} \) on line 15) and we do not analyze statements in a flow-sensitive manner (e.g. collapse control flow graph SCCs). However, none of these global variables need to be propagated into \( \text{proc} \) since they are all defined prior to being used by the call to \( \text{initGlobals} \) on line 8. An interprocedural flow-sensitive liveness analysis algorithm can conclude that this is the case by analyzing the definitions and uses of \( a \), \( b \), and \( c \) in a flow-sensitive manner; determining that none of these variables are live at the entry to \( \text{proc} \).

Algorithm 4.6 iterates over each procedure \( P \) in the program using a topological traversal of the acyclic call graph. A traversal over the control flow graph of \( P \) will
update two sets for each procedure \( (Q) \) that is reachable from \( P \):

**C\text{MOD}**: The set of SSA variables defined prior to some invocation of a procedure \( (Q) \). This set will be used to **constrain** the set of SSA variables passed into procedures.

**C\text{REF}**: The set of SSA variables used after some invocation of a procedure \( (Q) \). This set will be used to **constrain** the set of SSA variables passed out of procedures.

When the visited node \( N \) is an SCC (i.e. \( N \) contains more than one procedure or has a single recursive procedure), \( C\text{REF}[P] \) and \( C\text{MOD}[P] \) are identical for every procedure \( P \in N \), since \( P \) can be executed before and after every procedure in \( N \). Therefore, on line 4 we compute the set of SSA variables \( \text{SumREF} \) that can be used after every procedure in \( N \) exits by taking the union of:

1. SSA variables that are used in any given procedure within \( N \) (i.e. \( \text{REF}[Q] \) where \( Q \in N \)).

2. SSA variables that are used after any given procedure within \( N \) returns (i.e. \( C\text{REF}[Q] \) where \( Q \in N \)).

On the next line, we use a similar process to compute \( \text{SumMOD} \), which is the set of SSA variables that can be defined prior to entering a procedure in \( N \). Then, in the loop on lines 6–8 we set \( C\text{REF}[P] \) and \( C\text{MOD}[P] \) of each procedure \( P \in N \) to \( \text{SumREF} \) and \( \text{SumMOD} \), respectively. Moreover, in the following loop on lines 9–11, we add \( \text{SumREF} \) and \( \text{SumMOD} \) to the \( C\text{REF} \) and \( C\text{MOD} \) entries of each procedure called from \( N \), respectively.

If \( N \) is not an SCC, then it contains a single non-recursive procedure \( P \) and we apply a topological traversal over the acyclic CFG of \( P \) to update \( C\text{REF} \) and \( C\text{MOD} \) for procedures called from \( P \). During this traversal, we maintain \( \text{ProcSummary} \) and \( \text{ModSummary} \) which are the sets of reachable procedures and defined SSA variables,
**Algorithm 4.6** ISSA Liveness Analysis. For a procedure $P$, the set of SSA variables that may be used after $P$ exits is $C_{REF}[P]$ and the set of SSA variables that may be defined prior to invoking $P$ is $C_{MOD}[P]$.

**Input:** Acyclic Call Graph (ACG), $REF$, $MOD$

**Output:** $C_{REF}$ and $C_{MOD}$

1. $C_{REF} := C_{MOD} := \emptyset$
2. **foreach** node $N$ in a topological traversal over $ACG$ **do**
   3. **if** $|N| > 1$ or $N$ contains a recursive procedure **then**
      4. $Sum_{REF} := \bigcup_{Q \in N} REF[Q] \cup C_{REF}[Q]$
      5. $Sum_{MOD} := \bigcup_{Q \in N} MOD[Q] \cup C_{MOD}[Q]$
   6. **foreach** procedure $P \in N$ **do**
      7. $C_{MOD}[P] := Sum_{MOD}$
      8. $C_{REF}[P] := Sum_{REF}$
   9. **foreach** procedure $P \not\in N$ that is called from a procedure in $N$ **do**
      10. $C_{MOD}[P] := C_{MOD}[P] \cup Sum_{MOD}$
      11. $C_{REF}[P] := C_{REF}[P] \cup Sum_{REF}$
   12. **else** \{ $N$ contains a single non-recursive procedure $P$ \}
   13. $ModSummary := C_{MOD}[P]$, $ProcSummary := \emptyset$
14. **foreach** node $M$ in a topological traversal over the acyclic CFG of $P$ **do**
   15. **if** $M$ is an SCC **then**
      16. $NR := getUsedVars(M)$
      17. $ModSummary := ModSummary \cup getDefinedVars(M)$
      18. $NPC := getCalledProcs(M)$
      19. **foreach** procedure $Q \in NPC$ **do**
         20. $C_{MOD}[Q] := C_{MOD}[Q] \cup ModSummary$
         21. $ProcSummary := ProcSummary \cup NPC$
      22. **foreach** procedure $Q \in ProcSummary$ **do**
         23. $C_{REF}[Q] := C_{REF}[Q] \cup NR$
      24. **else** \{ $M$ contains a single basic block $BB$ \}
      25. **foreach** instruction $I \in BB$ **do**
         26. $NR := getUsedVars(I)$
         27. **foreach** procedure $Q \in ProcSummary$ **do**
            28. $C_{REF}[Q] := C_{REF}[Q] \cup NR$
         29. $NPC := getCalledProcs(I)$
         30. **foreach** procedure $Q \in NPC$ **do**
            31. $C_{MOD}[Q] := C_{MOD}[Q] \cup ModSummary$
            32. $ModSummary := ModSummary \cup getDefinedVars(I)$
            33. $ProcSummary := ProcSummary \cup NPC$
         34. **foreach** procedure $Q \in ProcSummary$ **do**
            35. $C_{REF}[Q] := C_{REF}[Q] \cup C_{REF}[P]$
respectively. Furthermore, we call three procedures that are passed region, which is either an instruction or an acyclic control flow graph node:

**getUsedVars**: Returns the set of SSA variables used within region.

**getDefinedVars**: Returns the set of SSA variables defined within region.

**getCalledProcs**: Returns the union of \( \mathcal{RPC}[ci] \ (\sum_{ci} \mathcal{RPC}[ci]) \), where \( ci \) is a call instruction within region.

The topological traversal processes each acyclic control flow graph node \( M \) on lines 14–33 in Algorithm 4.6. When \( M \) is an SCC, we must add ModSummary as well as all defined SSA variables to \( C\cdot\text{MOD}[Q] \) for each procedure \( Q \) that is called from \( M \). Moreover, we add each SSA variable used in \( M \) to the \( C\cdot\text{REF} \) entry of each procedure in ProcSummary as well as each procedure called from \( M \). Otherwise, \( M \) contains a single basic block \( BB \) that does not branch to itself and we proceed to visit each instruction \( I \) inside it. In this traversal, we add SSA variables used by \( I \) to \( C\cdot\text{REF} \) entries of procedures in ProcSummary and add ModSummary to \( C\cdot\text{MOD} \) entries of procedures called from \( I \).

In our implementation, calls to procedures getUsedVars, getDefinedVars, and getCalledProcs are merged into a single call that retrieves their corresponding sets (by applying a single traversal when a SCC is passed). For each call instruction \( ci \) we derive the set of procedures \( ci \) can reach \( \text{ReachProcs} = \mathcal{RPC}[ci] \) using the mapping \( \mathcal{RPC} \), which was presented in Section 2.4. Then, we compute the set of SSA variables used (getUsedVars) and defined (getDefinedVars) within a procedure \( P \in \text{ReachProcs} \) by querying the mappings \( \mathcal{REF} \) and \( \text{MOD} \), respectively.

**Example 4.3 ISSA Liveness Analysis for the example in Figure 3.1**

Please recall that \( \mathcal{REF}[B] = \emptyset, \mathcal{REF}[C] = \{x, y, z\}, \text{MOD}[B] = \{x\}, \) and \( \text{MOD}[C] = \emptyset \). Since the global variables \( y \) and \( z \) have an initializer, we will conclude that they are
defined prior to the invocation of every procedure. Since $x$ and $g$ are defined at the entry to procedure main on lines 11–12 in Figure 3.1(a), we can conclude that $x$ and $g$ are defined prior to $CL2$ and as such, $C\text{MOD}[C] = \{g, x, y, z\}$. Since $x$, $y$, and $z$ are used in procedure $C$, we can conclude that these variables are used after $CL1$ and as such, $C\text{REF}[B] = \{x, y, z\}$.

### 4.5.3 Pruning $\text{REF}$ and $\text{MOD}$

**Algorithm 4.7** Prune $\text{REF}$ and $\text{MOD}$ using $C\text{MOD}$ and $C\text{REF}$

**Input:** $\text{REF}$, $\text{MOD}$, $C\text{REF}$, and $C\text{MOD}$

**Output:** Pruned $\text{REF}$ and $\text{MOD}$

1. foreach procedure $P$ in the program do
2. $\text{MOD}[P] := \text{MOD}[P] \cap C\text{REF}[P]$
3. $\text{REF}[P] := \text{REF}[P] \cap C\text{MOD}[P]$

In Algorithm 4.7, we use $C\text{MOD}$ and $C\text{REF}$ to prune $\text{REF}$ and $\text{MOD}$, respectively. For each procedure $P$, $\text{MOD}[P]$ is constrained using the set of variables read after exiting $P$, while $\text{REF}[P]$ is constrained using the set of variables written prior to entering $P$.

As explained below, $\phi^V$ and $\phi^C$ instructions are inserted using $\text{REF}$ and $\text{MOD}$. Thus, pruning these sets reduces the number of $\phi^V$ and $\phi^C$ instructions that are inserted. Pruning $\text{REF}$ and $\text{MOD}$ by leveraging $C\text{REF}$ and $C\text{MOD}$ is similar to the use of the liveness analysis to reduce the insertion of redundant $\phi$ instructions during SSA form construction.

**Example 4.4** Pruning $\text{REF}$ and $\text{MOD}$ for the example in Figure 3.1

Since $\text{REF}[B] = \text{MOD}[C] = \emptyset$, we ignore them. The set $\text{REF}[C] = \{x, y, z\}$ will be constrained with $C\text{MOD}[C] = \{g, x, y, z\}$, however this will not remove any SSA variables from $\text{REF}[C]$. Similarly, $\text{MOD}[B] = \{x\}$ and $C\text{REF}[B] = \{x, y, z\}$ and as such, $\text{MOD}[B]$ will not change.
4.5.4 Inserting $\phi^V$ and $\phi^C$ Instructions

After $\mathcal{MOD}$ and $\mathcal{REF}$ are computed, we insert $\phi^V$ and $\phi^C$ instructions. Let us assume that $ci$ is a call instruction at the call site $cs$ in procedure $P$. Let us further assume that $ci$ can call a set of procedures $Targ(ci)$.

First, we describe how we propagate the values of SSA variables from $cs$ into a procedure $Q \in Targ(ci)$ by inserting $\phi^V$ instructions. We begin by computing the set of SSA variables used and defined in $Q$, which we refer to as $InVars := \mathcal{REF}[Q] \cup \mathcal{MOD}[Q]$. Then, for each SSA variable $var \in InVars$, we add the tuple $\langle cs, val \rangle$ to a $\phi^V$ instruction for $var$, which is located at the entry of procedure $Q$. The temporary $val$ holds the value of the load instruction $load \ @var$, which is placed right before $cs$. During $\phi$-placement and copy propagation these load instructions are replaced with the actual value of $var$ prior to $cs$. In addition, for each parameter $(par)$ of procedure $Q$ we add the tuple $\langle cs, arg \rangle$ to its $\phi^V$ instruction, where $arg$ is the argument of parameter $par$ at $cs$. Each SSA variable that does not have a $\phi^V$ instruction at the entry of a procedure is presumed undefined.

In order to propagate the values of SSA variables defined in a procedure $Q \in Targ(ci)$ into $P$ we insert $\phi^C$ instructions. Initially, we compute the set of SSA variables defined in $Targ(ci)$, which we refer to as $OutVars = \bigcup_{Q \in Targ(ci)} \mathcal{MOD}(Q)$. Afterwards, we create a $\phi^C$ instruction for each SSA variable $var \in OutVars$, which is located right after $cs$. For each procedure $Q \in Targ(ci)$, we add the tuple $\langle Q, val \rangle$ to this $\phi^C$ instruction, where $val$ is a temporary holding the value of a load instruction (placed at the end of procedure $Q$) whose pointer value is $var$. Moreover, if the return value of $ci$ is assigned to a temporary $%ci$, then we create a $\phi^C$ instruction that we assign to a temporary $%phic$ and proceed to replace uses of $%ci$ with $%phic$. For each procedure $Q \in Targ(ci)$ whose return value is equal to $rval$, we add $\langle Q, rval \rangle$ to the $\phi^C$ instruction whose result is assigned to $%phic$.

Example 4.5 Inserting $\phi^V$ and $\phi^C$ instructions for the examples in Figure 3.1 and Fig-
For Figure 3.1, we determined that $\mathcal{R} = \emptyset$ and $\mathcal{R} = \{x, y, z\}$, hence the SSA variables $x$, $y$, $z$ must be passed into procedure $C$ at $CI_2$. This is done by inserting the $\phi^V$ instructions on lines 5–7. The operands of these $\phi^V$ instructions were originally temporaries assigned loads of $x$, $y$, $z$ that were substituted during $\phi$ placement. Moreover, since $\text{MOD}(B) = \{x\}$ we propagate the value of $x$ at $CI_1$ using the $\phi^C$ instruction on line 15. A $\phi^V$ instruction that propagates $x$ into procedure $B$ was also inserted, but it is removed since the temporary holding the value that it ($\phi^V$ instruction) computes is not used.

In Figure 3.5(a), we do not have to propagate any SSA variables into or out of procedure $getPercentage$ because $\text{MOD}(getPercentage) = \mathcal{R} = \emptyset$. However, we insert a $\phi^V$ instruction for the parameters $%x$ and $%total$ on lines 3 and 5, respectively. Moreover, the two $\phi^C$ instructions on lines 22 and 24 propagate the return value of procedure $getPercentage$ at $CI_1$ and $CI_2$, respectively.

### 4.6 Interprocedural Copy Propagation

In Chapter 3, Section 3.4, we defined the value of a temporary, when it is used outside the procedure in which it is defined. This definition can be used to perform interprocedural copy propagation, because it enables us to fold certain $\phi^V$ and $\phi^C$ instructions. In this section, we outline the conditions that must be satisfied in order to fold an instruction to a given value. Moreover, we present an algorithm that folds $\phi^V$ and $\phi^C$ instructions.

As illustrated, in Section 3.4, we require additional guidelines to determine when it is legal to replace a named temporary with a given value. Our definition shows that it is legal to replace a temporary $%I0$ (whose defining instruction is $I$) with a value $V$ at a usage instruction $U$ when one of these conditions is satisfied:

1. $V$ is a constant. This includes numeric constants as well as the addresses of proce-
Chapter 4. Interprocedural Static Single Assignment Construction

Figure 4.3: Demonstrating why folding $\phi^V$ instructions merging a single value (prior to folding $\phi^C$ instructions) is legal.

dures and global variables.

2. $V$ is a temporary (whose defining instruction is $VI$ in procedure $Q$) and both of the following conditions are satisfied:

(a) None of the call instructions on any path between the program points of $I$ and $U$ can reach procedure $Q$.

(b) Either $I$ must not be a $\phi^C$ instruction or $P$ and $Q$ must not be in the same SCC. Otherwise, replacing $%I0$ with $V$ at any usage point of $%I0$ is illegal because $V$ would hold the value of an instance of $VI$ in the last call frame of $Q$ which is on the stack. However, $%I0$ is equal to the value of $VI$ in the last call frame of $Q$ which is popped off the stack.

In these scenarios, $V$ is identical at the program points of $I$ and $U$ under our definition and hence the replacement is legal.

If none of the temporaries assigned the result of $\phi^C$ instructions have been replaced (with a value), then we can replace any other temporary (i.e. not assigned the result of a $\phi^C$ instruction) without testing for the above conditions. In order to reason about this statement, we first explain why it is legal to replace $\phi^V$ instructions with the single value
they merge. Let us assume that the temporary $%I0$ holds the value of a $\phi^V$ instruction $I$, whose parent is procedure $P$. Let us further assume that $I$ can be folded to $%V0$, which holds the value of an instruction $V$ in procedure $Q$. Note that $V$ must dominate $I$ in order for such a replacement to be legal. Since none of the $\phi^C$ instructions have been folded, $%I0$ can only be used in procedure $P$ and its descendants. Hence, if there is a call instruction $CI$ that reaches $Q$ on a path between the program points of $I$ and a use of $%I0$ ($U$), then $P$ and $Q$ are in the same SCC. This scenario is illustrated in Figure 4.3(a). It is clear that procedure $Q$ can reach procedure $P$ because $V$ dominates $I$ and $U$ must be either in procedure $P$ or its descendants. While $CI$ reaches procedure $Q$, $%I0$ cannot be passed into procedure $Q$, because $V$ dominates $I$. Otherwise, as shown in Figure 4.3(b), $I$ would have to dominate $V$ and vice versa, which is impossible. Because $P$ and $Q$ are in the same SCC and $%I0$ cannot be passed into $Q$, the value of $%V0$ is the same at the program points of $I$ and $U$. This statement is true because:

If $P$ and $Q$ did not belong to the same SCC, then $P$ would not be able to reach procedure $Q$. Therefore, there would not be a call site that can reach procedure $Q$ (i.e. $CI$ in Figure 4.3(a)) on any path between the program points of $I$ and $U$. Otherwise, as previously discussed, $%I0$ cannot be passed into $Q$. Hence, the last call frame of $Q$ at the entry to procedure $P$ and at all usage points of $%I0$ is the same. Therefore, the value of $%V0$ at the program point of $I$ and at all uses of $%I0$ is the same.

As a result, it is legal to replace $%I0$ with $%V0$ at $U$. Moreover, the explanation above can be extrapolated to any other non-$\phi^C$ instruction that can be replaced with an instruction that dominates it.

### 4.6.1 Algorithm to Fold $\phi^V$ and $\phi^C$ Instructions

Once we begin replacing temporaries that are assigned the result computed by executing $\phi^C$ instructions, folding instructions becomes more complex, since replacing a temporary defined in one procedure with a temporary defined in another procedure may not be legal.
Because of this, we apply copy propagation in two steps. First, during $\phi$-placement we fold $\phi$, $\phi^S$, $\phi^L$, and $\phi^V$ instructions. Afterwards, we fold $\phi^C$ instructions that merge a single value.

In this section, we consider the replacement of a temporary $%I0 := \phi^C(\ldots, val)$, which holds the value of a $\phi^C$ instruction $I$ that merges a single value $val$. If $val$ is a constant then we can substitute $%I0$ with $val$ at all usage points without analyzing paths. Otherwise, we assume that $val$ is a temporary defined in procedure $Q$ and that $I$ corresponds to the call site $cs$. In order to replace $%I0$ with $val$ at a program point $U$, we must make sure that $val = %I0$ at $U$.

In Algorithm 4.8, we describe the replacement of temporaries, which hold the value of $\phi^C$ instructions located in procedure $P$, at usage points that are also located in procedure $P$. Conceptually, our algorithm constructs a virtual SSA form in a separate data structure, by creating a virtual SSA variable for each procedure. Algorithm 4.8 actually utilizes the iterated dominance frontier and applies a preorder traversal over the control flow graph. In our implementation, we maintain the values of virtual SSA variables in the mapping $\text{VirtVal}$ (i.e. $\text{VirtVal}[Q]$ is the value of the virtual SSA variable for procedure $Q$).

During the traversal, we visit instructions in procedure $P$ and replace $\phi^C$ instructions by analyzing the value of virtual SSA variables. Algorithm 4.8 guarantees that at a program point $U$, the value of $\text{VirtVal}[Q]$ will be equal to the propagation point of a temporary defined in procedure $Q$ at $U$. Otherwise, if a temporary in procedure $Q$ cannot be propagated to program point $U$, $\text{VirtVal}[Q]$ will be equal to $\emptyset$. We can substitute the temporary $%I0$ with $val$, if $\text{VirtVal}[Q] = cs$ when visiting $U$.

Because we are utilizing this virtual SSA form for replacing $\phi^C$ instructions, we focus on temporaries whose propagation points are call sites in $P$. Therefore, we only maintain the value of virtual SSA variables that correspond to procedures reachable from $P$. When $val$ is a temporary defined in a procedure that cannot be reached from $P$, then its
propagation point is the entry to $P$. Hence, we can substitute $\%I0$ with $val$ at all usage points of $\%I0$ within $P$.

**Algorithm 4.8** Replacing $\phi^C$ instructions at usage points. The input to this algorithm is the program and the mapping $VID$, that associates each basic block $BB$ with the set of procedures, whose virtual SSA variables have confluence points at the entry of $BB$. In addition, the input also consists of the mapping $RPC$.

1: `foreach` procedure $P$ do
2: \quad push($VisitStack$, $\langle$entry($P$), $\emptyset$ $\rangle$)
3: `while` !empty($VisitStack$) do
4: \quad $\langle BB, VirtVal \rangle := \text{pop}(VisitStack)$
5: \quad `repeat`
6: \quad \quad if NotVisited($BB$) then
7: \quad \quad \quad SetVisited($BB$)
8: \quad \quad `foreach` procedure $Q \in VID[BB]$ do
9: \quad \quad \quad VirtVal[$Q$] := $\emptyset$
10: \quad \quad `foreach` instruction $U$ in $BB$ from the entry of $BB$ do
11: \quad \quad \quad `foreach` operand $\%I0 = \phi^C(\ldots, val)$ of $U$ do
12: \quad \quad \quad \quad if $val$ is a constant then
13: \quad \quad \quad \quad \quad Replace $\%I0$ with $val$
14: \quad \quad \quad \quad else if $\%I0$ is defined in procedure $P$ then
15: \quad \quad \quad \quad \quad Let us assume that $val$ is a temporary defined in procedure $Q$
16: \quad \quad \quad \quad \quad Let us assume that $cs$ is the corresponding call site of $\%I0$
17: \quad \quad \quad \quad \quad if $Q$ and $P$ are not in the same SCC and VirtVal[$Q$] = $cs$ then
18: \quad \quad \quad \quad \quad Replace $\%I0$ with $val$
19: \quad \quad \quad \quad if $U$ is a call instruction at call site $cs$ then
20: \quad \quad \quad \quad \quad `foreach` procedure $Q \in RPC(U)$ do
21: \quad \quad \quad \quad \quad VirtVal[$Q$] := $cs$
22: \quad \quad \quad NextBB := $\emptyset$
23: \quad \quad `foreach` CFG successor of $BB$, $succ$ do
24: \quad \quad \quad if NotVisited($succ$) then
25: \quad \quad \quad \quad if NextBB = $\emptyset$ then
26: \quad \quad \quad \quad \quad NextBB := $succ$
27: \quad \quad \quad \quad else
28: \quad \quad \quad \quad \quad push($VisitStack$, $\langle$succ, VirtVal $\rangle$)
29: \quad \quad \quad $BB :=$ NextBB
30: \quad \quad until $BB$ = $\emptyset$

Now that we have provided an overview of Algorithm 4.8, we proceed to describe it in detail. Recall that we already computed $RPC$, which is a mapping that allows us to identify the procedures reached by each call instruction. At each call instruction $ci$,
whose call site is \( cs \), we let \( \text{VirtVal}[Q] := cs \) for each procedure \( Q \in \mathcal{RPC}[ci] \) (i.e. can be reached from \( ci \)). Given these assignments, a virtual SSA variable can have confluence points. Using the iterated dominance frontier we identify the confluence points of the virtual SSA variables and capture this in the mapping \( \mathcal{VID} \). The mapping \( \mathcal{VID} \) will associate each basic block with the set of procedures whose virtual SSA variables have confluence points at its entry.

After \( \mathcal{VID} \) is computed, we begin a preorder traversal of the control flow graph for \( P \), to copy propagate the virtual SSA variables. As stated, when reaching a call instruction at call site \( cs \), we assign \( cs \) to the virtual SSA variable of each reachable procedure. Note a temporary defined in \( Q \) can be propagated only through a single propagation point. However, two or more propagation points reach a confluence point of a virtual SSA variable. As such, when we visit a basic block \( BB \) we set \( \text{VirtVal}[Q] \) to \( \emptyset \) for each procedure \( Q \in \mathcal{VID}[BB] \) (on line 9 in Algorithm 4.8). Hence, each entry in \( \text{VirtVal} \) is equal to the propagation point of its corresponding procedure at the instruction we are currently visiting. As stated, this enables us to replace temporaries holding the value of \( \phi^C \) instructions.

To illustrate Algorithm 4.8, consider the program fragment shown in Figure 4.4(b). The call graph for this program is in Figure 4.4(a) and as it can be seen, procedures \( X \) and \( Y \) target procedure \( Z \). After the call site \( CI_1 \) in Figure 4.4(b) is visited, the virtual SSA variables for procedures \( X \) and \( Z \) are set to \( CI_1 \) by updating \( \text{VirtVal}[X] \) and \( \text{VirtVal}[Z] \). On line 21, in Algorithm 4.8, this step is taken because the call instruction at \( CI_1 \) can reach both procedures \( X \) and \( Z \). This will indicate that the propagation point of temporaries defined in procedures \( X \) and \( Z \) is \( CI_1 \) immediately after \( CI_1 \). After we visit the call site \( CI_2 \), we set the values of \( \text{VirtVal}[Y] \) and \( \text{VirtVal}[Z] \) to \( CI_2 \) and after \( CI_3 \), we set the value of \( \text{VirtVal}[Z] \) to \( CI_3 \). Since \( BB_2 \) is in the dominance frontier of the virtual SSA variables for procedures \( Y \) and \( Z \) (i.e. \( \mathcal{VID}[BB_2] = \{Y, Z\} \)), we set \( \text{VirtVal}[Y] \) and \( \text{VirtVal}[Z] \) to \( \emptyset \) at the entry to \( BB_2 \).
Figure 4.4: Example to illustrate the replacement of $\phi^C$ instructions using Algorithm 4.8. At the entry to $BB_2$, we set $\text{VirtVal}[Y]$ and $\text{VirtVal}[Z]$ to $\varnothing$ because $\mathcal{VID}[BB_2] = \{Y, Z\}$.

Example 4.6  Copy propagation in the examples shown in Figure 3.5 and Figure 3.1

In the ISSA form shown in Figure 3.1(b), the $\phi^C$ instruction defining $%x_1$ on line 15 can be replaced with $@z$, which is the address of variable $z$, since $@z$ is a constant.

In the ISSA form shown in Figure 3.5(a), $\text{VirtVal}[\text{getPercentage}]$ will be assigned $CI_1$ after line 21 and $CI_2$ after line 23. When reaching the instructions defining $%v_9$ and $%v_{10}$ on line 26 and line 27 in Figure 3.5(a), $\text{VirtVal}[\text{getPercentage}] = CI_2$. On line 26 and line 27 we use the temporaries $%v_6$ and $%v_7$, which hold the values of $\phi^C$ instructions that propagate the return values of procedure getPercentage at the call sites $CI_1$ and $CI_2$, respectively. Since $\text{VirtVal}[\text{getPercentage}] = CI_2$, $%v_7$ can be substituted with the return value from procedure getPercentage ($%v_{16}$) while $%v_6$ cannot.

4.7 Interprocedural Value Replacement

In this section, we describe our approach to testing whether or not it is legal to replace a temporary $%I_0 = I$ with a temporary $%J_0 = J$ at a usage of $%I_0$ (instruction $U$). We
assume that $I$ is in procedure $IP$, $J$ is in procedure $JP$, and that the instruction $U$ uses $%I0$ and is in procedure $UP$. Moreover, we assume that either $I$ is not a $\phi^C$ instruction or that $IP$ and $JP$ are not in the same SCC.

Given the assumptions above, we can replace $%I0$ with $%J0$ at $U$, if $JP$ is not invoked between the execution of $I$ and $U$. This condition is satisfied if $JP$ dominates $IP$. We reason about this statement by examining two situations. On the one hand, if $JP$ reaches $UP$, then $%J0$ will hold a value defined in the same call frame of $JP$ at the program points of both $I$ and $U$. This is because $%I0$ (which is replaced with $%J0$) cannot be propagated into $JP$ as was already illustrated in Figure 4.3 and explained in Section 4.6. On the other hand, if $JP$ does not reach $UP$, then both $%I0$ and $%J0$ hold a value defined in a call frame that was popped off the stack. Once $%I0$ is assigned the (result computed by the) instance of $I$ that will be used at the program point $U$, $JP$ cannot be invoked by any call instruction until $U$ is executed. If such a call instruction did exist, it would reach procedure $IP$ as well and as such, $%I0$ would hold a different value.

In this section, we focus on the scenario where $JP$ does not dominate $IP$. In this scenario, we compute the call graph paths to the last invocation of $JP$ at the program points of $I$ and $U$. If the path to the last invocation of $JP$ at $I$ is a postfix of the path to the last invocation of $JP$ at $U$, then $JP$ is not invoked between the execution of $I$ and $U$ and we can replace $%I0$ with $%J0$ at $U$. In order to replace such temporaries using this approach, we leverage the interprocedural value replacement map, which we refer to as $IVR$.

### 4.7.1 Testing Interprocedural Value Equality

Conceptually, for a given procedure $JP$, $IVR$ maintains a mapping between certain program points and the call graph path to $JP$’s last invocation. The program points where this mapping is maintained are call sites, procedure entries, and confluence points.
Algorithm 4.9 Identify the entry within $\mathcal{IVR}$ that contains the last call graph path to procedure $P$ at instruction $I$. We assume that $\mathcal{IVR}[P]$ contains the entries $Ent_1, \ldots, Ent_N$ within the parent procedure of instruction $I$.

1: $FirstDom := \emptyset$
2: for $i := 1$ to $N$ do
3: \hspace{1em} if $Ent_i \neq I$ and $Ent_i$ dominates $I$ and ($FirstDom = \emptyset$ or $FirstDom$ dominates $Ent_i$) then
4: \hspace{2em} $FirstDom := Ent_i$
5: return $FirstDom$

of the virtual SSA variables in Algorithm 4.8. At these confluence points, more than one call graph path reaches the last invocation of procedure $JP$.

In order to identify the entry in $\mathcal{IVR}[JP]$ that contains the call graph path to the last invocation of $JP$ at a program point, we apply Algorithm 4.9. Algorithm 4.9 iterates through all the entries in $\mathcal{IVR}[JP]$ located in the parent procedure of $U$ and returns the immediate dominating entry of $U$, which dominates $U$ but does not dominate any other entry that also dominates $U$. In this manner, we identify $EntI$ and $EntU$ which are the immediate dominating entries for $I$ and $U$ in $\mathcal{IVR}[JP]$, respectively. The value of $\%J0$ is identical at the program points of $I$ and $U$ if $\mathcal{IVR}[JP][EntI]$ is a postfix of the path $\mathcal{IVR}[JP][EntU]$.

Entries in $\mathcal{IVR}$ for one procedure can sometimes be reused. Let us assume that procedure $Q$ dominates procedure $P$ and that $P$ and $Q$ are not in the same SCC. In this case, each call graph path reaching $P$ must pass through $Q$. Hence, the call graph path reaching the last invocation of $Q$ is a prefix of the call graph path reaching the last invocation of $P$, in all procedures that are not reachable from $Q$. Therefore, we reuse the mapping for $Q$ to compute the last call graph path reaching $P$, at all program points that are not reachable from $Q$. This saves us additional memory space, since a number of entries do not have to be saved or propagated. To simplify the presentation, we assume that $\mathcal{IVR}$ contains entries for each procedure at call sites and procedure entries.

Example 4.7 Leveraging call graph paths to determine value propagation legality

Consider Figure 4.5 where we present the call graph paths to the last invocation of pro-
Chapter 4. Interprocedural Static Single Assignment Construction

procedure foo3 in the basic blocks BB1 and BB3.

Let us consider an instruction whose result is assigned to \( \%x2 \) in procedure foo1 that is used in BB1 and can be folded to the temporary \( \%x1 \), which is defined in procedure foo3. In the basic block BB1, the call graph path to the last invocation of foo3 is \( \langle CI1, CI2, CI3 \rangle \). We can replace \( \%x2 \) with \( \%x1 \) in BB1, because the call graph path to the last invocation of foo3 at the program point where \( \%x2 \) is defined (\( \langle CI2, CI3 \rangle \)) is a postfix of the call graph path to the last invocation of foo3 in BB1.

If \( \%x2 \) was used in BB3, this replacement would be illegal since in BB3 the call graph paths to the last invocations of foo3 is \( \langle CI4, CI3 \rangle \). Since \( \langle CI2, CI3 \rangle \) is not a postfix of \( \langle CI4, CI3 \rangle \) we conclude that \( \%x1 \) is not equal to \( \%x2 \) in BB3.

Note that the full path to the last invocation of foo3 is not required to test whether \( \%x2 \) can be replaced with \( \%x1 \) in BB1 and BB3. In the above tests, CI3 is common to all paths and removing it has no impact on the result. This is because procedure foo2
dominates foo3 and all the tested program points are not reachable from foo2. We utilize this property to save memory space in \( \mathcal{IVR} \) (reduce the number of entries).

### 4.7.2 Computing the Interprocedural Value Replacement Map

In order to compute \( \mathcal{IVR} \), we use a topological traversal over the acyclic call graph. As illustrated in Figure 4.6, the entry procedure is visited first, followed by procedures \( A, B \), and the collapsed SCC, which contains procedures \( C \) and \( D \). During the IR traversal we update \( \mathcal{IVR} \) with the values of virtual SSA variables. Let us assume that we currently visit a call instruction \( ci \) (whose call site is \( cs \)) that targets a single procedure \( T \) (i.e. \( \text{Targ}(ci) = \{T\} \)) and can reach each procedure in the set \( \text{ReachProcs} = \mathcal{RPC}[ci] \). In this case, we first update the entry for \( T \) in \( \mathcal{IVR} \) with the current values of the virtual SSA variables. This is illustrated in Figure 4.6(b) where \( \text{VirtVal}[A] \) is equal to \( CI1 \), just before the call site \( CI2 \). It can be observed that \( \text{VirtVal}[A] \) is propagated to procedure \( B \) at \( CI2 \) by setting \( \mathcal{IVR}[A][B] := CI1, \ldots \) to its value (i.e. the call site \( CI1 \)). Afterwards, we update the values of virtual SSA variables for procedures in the set \( \text{ReachProcs} \). For each procedure \( Q \in \text{ReachProcs} \) we set the value of \( \text{VirtVal}[Q] \) and \( \mathcal{IVR}[Q][cs] \) to \( cs \).
When a confluence point $BB$ is encountered for a virtual SSA variable corresponding to procedure $Q$, we add the incoming value of $VirtVal[Q]$ (from the predecessor) to $IVR[Q][BB]$.

When visiting the instructions in a procedure, we update $IVR$ entries with the call site that corresponds to the last invocation of a procedure as opposed to the call graph path. It may appear that the call graph path associated with these call sites is still unavailable. However, we are able to derive call graph paths through which values can be propagated by analyzing call sites. Let us assume that the propagation point of a temporary $%I0$, which is defined in procedure $P$, is at a call instruction $ci$ in procedure $Q$. Note that $%J0$ can be propagated out of $Q$ only if the three conditions below are met:

1. Procedure $Q$ terminates and as such it has an exit node $Exit$.

2. There is no path in the control flow graph of $Q$ between $ci$ and another call instruction reaching $P$.

3. The call instruction $ci$ dominates $Exit$.

We illustrate this using an example in Figure 4.7(a). Note that a temporary defined in procedure $P$ cannot be propagated out of procedure $Q$ because the exit node is a confluence point for the virtual SSA variable that corresponds to procedure $P$. In other words, multiple call sites could correspond to the propagation point of a temporary defined in $P$ at the exit node. In addition, such a propagation is impossible because the call instruction at the call site $CI$ does not dominate the exit node.

In order for a temporary defined in $P$ to be propagated out of $Q$, the call graph path of the last invocation of $P$ must pass through the same call site in $Q$ (which matches the conditions outlined above). We refer to these call sites as the ending call sites of $P$. Note that the call graph paths to the last invocation of procedure $P$ are composed of the ending call sites of $P$. In order to derive the call graph path to $P$’s last invocation at a
Chapter 4. Interprocedural Static Single Assignment Construction

(a) Invalid propagation example. (b) Diagram illustrating the call relation between procedures and the control flow relation between call sites within the same procedure. (c) Ending call site graph for procedure A.

Figure 4.7: Examples illustrating how IVR derives relevant call graph paths from call sites.

given call site, we leverage the ending call site graph of P. In the ending call site graph of P, an edge is constructed between each ending call site cs in procedure Q and the call sites that target Q.

The ending call site graph is derived using a traversal that starts at procedure P and moves up the call graph to its predecessors. The traversal will stop when we reach the entry procedure or find a predecessor that either dominates P or cannot propagate instructions whose parent procedure is P; this situation is illustrated in Figure 4.7(a) and the process is illustrated in Figure 4.7(b). In this example, all call sites within a given procedure have the same callee. For instance, procedure A is called at CI1 and CI2 since CI1 and CI2 are located in procedure B and there is an arrow from B to A.

To identify the call sites on a path through which a value is propagated out of procedure A, we derive the ending call site graph for procedure A, which is shown in Figure 4.7(c). Note that CI2 is the ending call site of procedure A in procedure B and procedure B is targeted by call sites CI3, CI4, and CI5. Hence, an instruction in procedure A that is propagated out of CI3, CI4, or CI5 must also be propagated out of CI2. This is captured in the ending call site graph with edges between these nodes. For the same reason, an edge is inserted between CI5 and CI6 in Figure 4.7(c).

We can determine whether the value produced in procedure P is equal at two program
points whose $IVR$ entries map to call sites $CI1$ and $CI2$ by applying a traversal over the ending call site graph of $P$. If $CI1$ is an ancestor of $CI2$ in the ending call site graph of $P$, or vice versa, then these values are equal.

**Example 4.8** Leveraging $IVR$ to determine value propagation legality

Consider Figure 4.8 which contains the ending call site graph for procedure $foo3$ in Figure 4.5. In this example, we repeat the legality test in Example 4.7 by using $IVR$ to determine whether we can replace $%x2$ with $%x1$ at usage points of $%x2$.

Note that the entry in $IVR[foo3]$ for the definition site of $%x2$ is $CI2$. Moreover, the entry in $IVR[foo3]$ for the uses of $%x2$ in BB1 and BB3 are $CI1$ and $CI4$, respectively. Since $CI1$ is a descendant of $CI2$ in the ending call site graph of $foo3$, we can replace $%x2$ with $%x1$ in BB1. However, because $CI4$ is not a descendant of $CI2$ in the ending call site graph of $foo3$, we cannot replace $%x2$ with $%x1$ in BB3.

### 4.8 Experimental Evaluation

In this section, we evaluate the proposed ISSA construction algorithm. Previous work [32, 45] uses field-insensitive pointer analysis algorithms [?, ?], does not perform either inter-procedural copy propagation or ISSA liveness analysis, and only includes scalar globals in the set of SSA variables. To quantify the benefit of our techniques over previous work,
we implemented the proposed algorithms to construct ISSA form in the LLVM [30] compiler infrastructure as a sequence of passes in the optimizer. Prior to constructing ISSA form, we convert the IR to SSA form, and perform constant propagation (-ipconstprop,-instcombine,-sccp) as well as dead code removal (-adce, -dce). Afterwards, we use the field-sensitive (context-insensitive and flow-insensitive) pointer analysis outlined in Section 4.2 to derive the point-to graph and construct ISSA form as outlined in this chapter.

Testing was done by comparing the output of the ISSA executable with the reference (i.e. GCC output) and through a number of sanity checks that verified various IR properties. In order to generate an executable, we performed an out-of-ISSA translation (using naive algorithm and advanced algorithm, presented in Chapter 5) and applied the usual compiler passes afterwards.

To evaluate the proposed techniques, we used the MediaBench [31] and a set of SPECINT2000 [1] benchmarks. In Table 4.1, we list the various benchmarks used, their lines of code, and the number of call sites present in those benchmarks. The experiments were performed on an Intel CORE 2 Duo 1.66 GHz processor with 4 GB memory and running 64-bit Ubuntu. The ISSA form was generated from SSA form, after constant propagation and dead code removal were applied.

Table 4.1: Benchmark characteristics and the time it takes to construct ISSA (column labeled Time), in seconds.

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines Of Code</th>
<th>Call Sites</th>
<th>Time (s)</th>
<th>Name</th>
<th>Lines Of Code</th>
<th>Call Sites</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>G721</td>
<td>1476</td>
<td>53</td>
<td>0.30</td>
<td>164.gzip</td>
<td>8218</td>
<td>306</td>
<td>0.95</td>
</tr>
<tr>
<td>GSM</td>
<td>4626</td>
<td>258</td>
<td>1.40</td>
<td>175.vpr</td>
<td>16984</td>
<td>1902</td>
<td>5.88</td>
</tr>
<tr>
<td>JPEG</td>
<td>26173</td>
<td>942</td>
<td>10.80</td>
<td>181.mcf</td>
<td>1913</td>
<td>81</td>
<td>1.02</td>
</tr>
<tr>
<td>MPEG2</td>
<td>7283</td>
<td>654</td>
<td>2.30</td>
<td>186.crafty</td>
<td>19478</td>
<td>2252</td>
<td>8.32</td>
</tr>
<tr>
<td>197.parser</td>
<td>10932</td>
<td>1691</td>
<td>21.52</td>
<td>254.gap</td>
<td>59493</td>
<td>9773</td>
<td>91.17</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4665</td>
<td>299</td>
<td>0.74</td>
<td>300.twolf</td>
<td>19756</td>
<td>1883</td>
<td>38.63</td>
</tr>
</tbody>
</table>
We also present the runtime for ISSA generation in Table 4.1. All the MediaBench [31] benchmarks complete within a few seconds. The runtime is longer for the SPECINT2000 [1] benchmarks. However, this is expected, as the benchmarks usually have more lines of code and a greater number of call sites. Furthermore, SPECINT2000 benchmarks use a greater set of C language features, including recursion, indirect calls, and cast accesses, which increase the number of SSA variables that need to be passed in and out of procedures.

4.8.1 Scalability

We do not include the benchmarks 255.vortex and 176.gcc from SPECINT2000 in our study, because our ISSA construction algorithm currently does not scale to them. This is mainly due to the space consumed by $\phi$ and $\phi^V$ instructions.

![Figure 4.9: Number of $\phi$ instructions inserted in the benchmarks 255.vortex and 176.gcc as we process methods during $\phi$ placement.](image)

Figure 4.9: Number of $\phi$ instructions inserted in the benchmarks 255.vortex and 176.gcc as we process methods during $\phi$ placement.

In Figure 4.9, we present the number of $\phi$ instructions inserted during $\phi$ placement as we iterate through the procedures of the benchmark. As the number of $\phi$ instructions increased in these benchmarks, the system ran out of memory space and eventually

\^decoder
crashed. In Figure 4.9 for the benchmark 255.vortex, we show the number of \( \phi \) instructions inserted in the first 309 of 963 procedures. While the SSA form IR contains just 2564 \( \phi \) instructions, during ISSA construction the number increases to over 3.5 million after a third of the program is processed. Shortly after processing 309 procedures the program crashes. We observed a similar problem with the benchmark 176.gcc, which has over 2000 procedures. After processing over 250 procedures, the ISSA form construction increased the number of \( \phi \) instructions from 14627 to roughly 3 million.

One reason that we observed the large increase in the number of \( \phi \) instructions is because the impact of a definition is extended to the whole program. In other words, an assignment in a procedure \( Q \) may result in the insertion of \( \phi \) instructions in procedures that reach \( Q \). Moreover, at store and load instructions processed by Algorithm 4.3, the average number of SSA variables that may be accessed is 49 and 38 for 255.vortex and 176.gcc, respectively. This is significantly higher than other benchmarks and as indicated in Figure 4.10, a vast number of store and load instructions can access over 50 SSA variables.

An increase in the number of accessed variables will result in additional \( \phi \) instructions,
since more definitions must be propagated to a greater set of program points. When analyzing a number of these store and load instructions, we noted that the pointer value can only access one or a few of the SSA variables at runtime. Hence, improving the precision of the pointer analysis may lower the number of accessed SSA variables which will reduce the number of $\phi$ instructions.

### 4.8.2 Excluded Benchmarks

In addition to the benchmarks 176.gcc and 255.vortex from SPECINT2000 we do not include the following benchmarks:

**252.eon:** This is a C++ benchmark, and the current implementation does not handle various C++ features.

**253.perlbmk:** When running the benchmark, a segmentation fault occurred (in baseline LLVM and GCC) which seemed to be caused by a 64-bit pointer issue.

### 4.8.3 Impact of Increasing the Scope and Resolution

We now evaluate the impact of increasing the scope and resolution of ISSA using the number of SSA variables and the number of load instructions resolved to the corresponding definition. A greater number can provide a greater benefit to clients of ISSA.

One improvement of our work over the ISSA construction algorithms of Staiger et al. [45] and Liao [32] is that we include structure fields and singleton heap variables in the set of SSA variables. In Table 4.2, we evaluate the impact of extending the set of SSA variables to include singleton heap variables and structure fields in addition to scalar globals.

As indicated, we have 5.3 times more SSA variables than previous work [32,45]. The highest improvement was observed in the benchmark JPEG, for two reasons. First, the set of SSA variables is extended to include fields within structures allocated on the stack.
Table 4.2: Number of SSA variables, load instructions replaced, and singular allocation sites identified.

in procedure *main*. Second, the benchmark *JPEG* allocates a structure on the heap for each file type that it converts (to .jpeg format). Since these allocation instructions are executed once at most, we include their fields in the set of SSA variables. The large and consistent increase in the set of SSA variables demonstrates that our proposed techniques significantly increase the set of SSA variables, thus making ISSA more useful.

In Table 4.2, we also quantify the impact of increasing the set of SSA variables on the number of load instructions substituted with a value (see columns underneath *Loads Replaced*). On average, we substituted 2.2 times more load instructions than an ISSA construction similar to Staiger et al. [45]. This means that compiler analyses and optimizations can leverage our ISSA form to identify the reachable definitions of more program variable uses. Therefore, the potential benefit to applications of ISSA is increased by using our approach.

Lastly, we present the number of singular allocation sites in the benchmarks. While a large percentage of singular allocation sites were identified in a number of benchmarks,
this translated to a substantial increase in SSA variables only in the benchmark JPEG. In other benchmarks, singular allocation sites were primarily used for arrays, which we currently do not include in the set of SSA variables.

### 4.8.4 Impact of Copy Propagation and ISSA Liveness Analysis

We compute the sum of $\phi_V$ and $\phi_C$ instructions to evaluate the impact of copy propagation and ISSA liveness analysis. The sum $\phi_V$ and $\phi_C$ instructions is a good metric for a number of reasons. First, a lower number of $\phi_V$ and $\phi_C$ instructions will indicate that less memory is consumed by the ISSA representation of the program. Moreover, reducing the number of $\phi_V$ and $\phi_C$ instructions during ISSA form construction will result in less computation as well, since fewer values will have to be copy propagated and fewer $\phi$ instructions will be placed. In addition, folding $\phi_V$ and $\phi_C$ instructions improves precision, since we eliminate false merge points. Hence, fewer $\phi_V$ and $\phi_C$ instructions will reflect improvement both in terms of the ISSA-construction performance and precision.

We apply copy propagation as described in Section 4.6 and fold $\phi_V$ and $\phi_C$ instructions. As shown in Table 4.3, copy propagation reduced the number of $\phi_V$ and $\phi_C$ instructions at call sites and procedure entries by 44.5% on average. In addition, during copy propagation, we folded 30% of the $\phi_V$ instructions that were inserted as well as a number of $\phi^L$ and $\phi^S$ instructions. This demonstrates a significant improvement over previous work, which did not perform interprocedural copy propagation.

In Table 4.3, we detail the impact of the ISSA liveness analysis presented in Section 4.5.2, on reducing the number of $\phi_V$ and $\phi_C$ instructions. The second and third columns contain the sum of $\phi_V$ and $\phi_C$ instructions without and with using the ISSA liveness analysis to prune the read and write sets, respectively. On average, 24.8% of the $\phi_V$ and $\phi_C$ instructions were removed using the ISSA liveness analysis, demonstrating the benefit of our proposed algorithm.

While a more sophisticated algorithm can enable us to further reduce the number of
### Table 4.3: Impact of ISSA liveness analysis and copy propagation measured by the reduction in the number of $\phi^V$ and $\phi^C$ instructions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ISSA Liveness Analysis</th>
<th>Copy Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\phi^V, \phi^C$</td>
<td>$\phi^V, \phi^C$</td>
</tr>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>G721</td>
<td>133</td>
<td>100</td>
</tr>
<tr>
<td>GSM</td>
<td>494</td>
<td>319</td>
</tr>
<tr>
<td>JPEG</td>
<td>10261</td>
<td>9115</td>
</tr>
<tr>
<td>MPEG2</td>
<td>6279</td>
<td>5408</td>
</tr>
<tr>
<td>164.gzip</td>
<td>2606</td>
<td>2074</td>
</tr>
<tr>
<td>175.vpr</td>
<td>7864</td>
<td>4734</td>
</tr>
<tr>
<td>181.mcf</td>
<td>262</td>
<td>181</td>
</tr>
<tr>
<td>186.crafty</td>
<td>20935</td>
<td>16373</td>
</tr>
<tr>
<td>197.parser</td>
<td>23037</td>
<td>22015</td>
</tr>
<tr>
<td>254.gap</td>
<td>100678</td>
<td>61684</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>942</td>
<td>614</td>
</tr>
<tr>
<td>300.twolf</td>
<td>5211</td>
<td>4106</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

φ$^V$ and φ$^C$ that are inserted, we noted that the benefit is limited. Using copy propagation, we can remove instructions defining temporaries that are not needed. When doing this, we noted that the additional $\phi^V$ and $\phi^C$ instructions that were removed (indicating the maximum benefit that can be derived from more sophisticated analysis) ranged from 10% to 20%.

#### 4.8.5 Impact of Pointer Analysis

The point-to graph is computed by using an inclusion-based and field-sensitive pointer analysis [36, 53], which is more precise than the pointer analyses used in previous ISSA construction algorithms [32, 45]. This will reduce the number of $\phi, \phi^V, \phi^C, \phi^S$, and $\phi^L$ instructions in the resulting ISSA form.

In Table 4.4, we illustrate the difference between the number of SSA variables that need to be propagated into and out of procedures when using the field-insensitive pointer analysis available in LLVM and the field-sensitive pointer analysis. As indicated, when
Table 4.4: Size of $\mathcal{REF}$ and $\mathcal{MOD}$ when generating ISSA with a field-insensitive and field-sensitive pointer analysis.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Field-Sensitive</th>
<th>Field-Insensitive</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>G721</td>
<td>10</td>
<td>83</td>
<td>8.3</td>
</tr>
<tr>
<td>GSM</td>
<td>214</td>
<td>818</td>
<td>3.8</td>
</tr>
<tr>
<td>JPEG</td>
<td>330</td>
<td>2480</td>
<td>7.5</td>
</tr>
<tr>
<td>MPEG2</td>
<td>1256</td>
<td>12185</td>
<td>9.7</td>
</tr>
<tr>
<td>164.gzip</td>
<td>1024</td>
<td>4348</td>
<td>4.3</td>
</tr>
<tr>
<td>175.vpr</td>
<td>2265</td>
<td>18341</td>
<td>8.1</td>
</tr>
<tr>
<td>181.mcf</td>
<td>49</td>
<td>136</td>
<td>2.8</td>
</tr>
<tr>
<td>186.crafty</td>
<td>2660</td>
<td>11236</td>
<td>4.2</td>
</tr>
<tr>
<td>197.vpr</td>
<td>8239</td>
<td>21398</td>
<td>2.6</td>
</tr>
<tr>
<td>300.twolf</td>
<td>581</td>
<td>40806</td>
<td>70.2</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>12.2</td>
</tr>
</tbody>
</table>

using the field-insensitive pointer analysis, more SSA variables have to be propagated into and out of procedures. For the benchmark 300.twolf, we observed a drastic decrease in the number of SSA variables because the universal object in the field-insensitive pointer analysis was aliased to a large number of SSA variables. Because the pointer values of many load and store instructions can point to the universal object, the SSA variables aliased with the universal object had to be propagated into or out of a large number of procedures.

The number of SSA variables propagated into and out of procedures is, on average, 12.2 times higher in the field-insensitive version, primarily because of the greater point-to-set size. Furthermore, since the pointer analysis is used to identify the procedures that may be invoked by indirect calls, the call graph of the field-insensitive version usually contains spurious paths. This increases the size of the input and output sets, as data is propagated across additional (and unreachable) procedures. Larger sets result in increased code size and runtime. Hence, by using the field-sensitive pointer analysis, we are able to reduce code size and runtime, in addition to being able to increase the set of SSA variables.
### Table 4.5: Numerical summary of the data in Figure 4.11, which includes the percentage and relative space consumption of the new ISSA instructions.

<table>
<thead>
<tr>
<th></th>
<th>$\phi^L$</th>
<th>$\phi^S$</th>
<th>$\phi^V$</th>
<th>$\phi^C$</th>
<th>$\phi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>1.24%</td>
<td>1.38%</td>
<td>16.24%</td>
<td>30.84%</td>
<td>50.31%</td>
</tr>
<tr>
<td>Space consumed</td>
<td>1.72%</td>
<td>1.55%</td>
<td>30.03%</td>
<td>12.35%</td>
<td>54.35%</td>
</tr>
</tbody>
</table>

### 4.8.6 ISSA IR in Benchmarks

In Figure 4.11, we provide more detail regarding the newly inserted ISSA instructions. In Figure 4.11(a), we examine the IR and provide the percentage of $\phi^L$, $\phi^S$, $\phi^V$, $\phi^C$, and $\phi$ instructions that are inserted. In Figure 4.11(b), we provide the relative size consumed by these instructions. A numerical summary of the above figures is provided in Table 4.5.

It is clear that the $\phi^S$ and $\phi^L$ instructions are far less frequent and consume less space than other kinds of instructions. This demonstrates that capturing the impact of conditional load and store instructions on SSA variables can be handled very efficiently using $\phi^S$ and $\phi^L$ instructions. In contrast, $\phi^V$ and $\phi^C$ instructions consumed more space. While more $\phi^C$ instructions were inserted, they tended to have a single operand. However, $\phi^V$ instructions usually merged values from multiple call sites and as such, consumed more space.

The largest increase was due to the insertion of $\phi$ instructions. Note that due to a single assignment (to an SSA variable) in procedure $P$, we may have to insert $\phi$ instructions in all predecessors of $P$. As such, the impact of these assignments extends to the whole program and in fact, most of the $\phi$ instructions were inserted because we had to account for assignments to SSA variables at call sites.

In Figure 4.12, we present the percentage of ISSA IR in benchmarks and the space it occupies. In a number of benchmarks, where copy propagation performed well, the percentage of ISSA instructions is quite small. By folding $\phi^V$ and $\phi^C$ instructions, copy propagation reduces ISSA instructions directly and also eliminates false merge points. In turn, this enables us to fold additional $\phi$ instructions. In the benchmarks 186.crafty
(a) The percentage of $\phi^L$, $\phi^S$, $\phi^V$, $\phi^C$, and $\phi$ instructions.

(b) The percentage of memory space occupied by $\phi^L$, $\phi^S$, $\phi^V$, $\phi^C$, and $\phi$ instructions.

Figure 4.11: The percentage of $\phi^L$, $\phi^S$, $\phi^V$, $\phi^C$, and $\phi$ instructions as well as the space they occupy. Space consumption is computed by adding the number of instructions and their operands.
Figure 4.12: The percentage of $\phi^L$, $\phi^S$, $\phi^V$, $\phi^C$, and $\phi$ instructions and the memory space they occupy in relation to all other instructions.

and 197.parser, copy propagation was less effective due to recursive procedures, as $\phi^C$ instructions that propagate temporaries defined in the same call graph SCC cannot be folded.

4.8.7 Library Calls

As mentioned in Section 4.4, load and store instructions may be inserted around call instructions that invoke library procedures. In Table 4.6, we present the number of load and store instructions that are inserted around these call instructions. In our implementation, we accounted for the impact of calls to common libc functions by identifying the arguments that pass references and then leveraged the pointer analysis to determine the SSA variables that may be modified or used by these call instructions. This resulted in fewer load and store instructions being inserted. In fact, as illustrated in Table 4.6, the number of load and store instructions that were inserted is relatively small. Moreover, the out-of-ISSA translation (presented in Chapter 5) will remove all store instructions inserted due to library calls and every redundant load instruction.
Table 4.6: The number of load and store instructions inserted to write and retrieve the value of SSA variables around call instructions invoking library procedures.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Load Instructions</th>
<th>Store Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSM</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>MPEG2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>G721</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>JPEG</td>
<td>66</td>
<td>67</td>
</tr>
<tr>
<td>164.gzip</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>175.vpr</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>181.mcf</td>
<td>28</td>
<td>35</td>
</tr>
<tr>
<td>186.crafty</td>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>197.parser</td>
<td>75</td>
<td>17</td>
</tr>
<tr>
<td>254.gap</td>
<td>23</td>
<td>29</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0</td>
<td>172</td>
</tr>
</tbody>
</table>

4.9 Summary

This chapter presents and evaluates an algorithm to construct ISSA. We have shown that while handling a large number of SSA variables, we are still able to construct ISSA in seconds. ISSA improves precision by handling a large percentage of load instructions and by resolving a few pointer dereferences. We also demonstrated that an interprocedural live variable and an undefined variable analysis can be leveraged to reduce the insertion of redundant $\phi^V$ and $\phi^C$ instructions. Moreover, we have demonstrated that our copy propagation algorithm can replace and then remove a significant number of $\phi^V$ and $\phi^C$ instructions.
Chapter 5

Out-of-ISSA Translation

5.1 Introduction

A natural step towards integrating ISSA into a compiler is to convert the IR back to SSA form, a process referred to as out-of-ISSA translation, which is the focus of this chapter. While out-of-SSA translation algorithms have been previously proposed [9,11,18,39,44], we found that performance is degraded if we naively extend these algorithms to translate out of ISSA form. In this chapter, we present an out-of-ISSA translation algorithm and a storage-remap transformation that improve the performance of the code.

The chapter is organized as follows. Section 5.2 reviews the literature on out-of-SSA translation and discusses the additional challenges and opportunities in an out-of-ISSA translation. This section also introduces important terminology used in later sections and a running example that will be used to illustrate our out-of-ISSA translation. In Section 5.3, we describe the storage-remap transformation and detail the other passes applied besides it. In Section 5.4, we present our proposed out-of-ISSA translation algorithm. In Section 5.5, we present an experimental study and the associated results. Finally, a summary is provided in Section 5.6.
5.2 Background and Related Work

5.2.1 Out-of-SSA Translation

Over the years, out-of-SSA translation algorithms have been refined and improved. Proposed by Cytron [18], the first out-of-SSA translation algorithm replaced each k-input \( \phi \) instruction with \( k \) copy instructions; one at the end of each predecessor basic block. Consider the example shown in Figure 5.1. In this example, \( %v1 \) is equal to 20 or 30, if entering \( \text{MergeBB} \) from \( BB0 \) or \( BB1 \), respectively. When applying Cytron’s out-of-SSA translation algorithm [18] we first allocate the scalar stack variable whose address is held in \( %var \). Then, at the end of the basic block \( BB0 %var \) is assigned 20 and at the end of \( BB1 \) it is assigned 30. References to \( %v1 \) are replaced with loads of \( %var \) (as was done in the basic block \( UseBB \)), thus allowing us to erase \( %v1 \).

Briggs et al. [11] identified two problems with Cytron’s algorithm due to parallel copies and critical edges in the control flow graph and proposed a revised out-of-SSA translation algorithm that addresses these problems. Sreedhar et al. [44] proposed a more comprehensive solution. In contrast to Cytron et al. [18], an additional variable is allocated and another store instruction is placed prior to each replaced \( \phi \) instruction. Using the algorithm proposed by Sreedhar et al. [44], the out-of-SSA translation, shown in Figure 5.1(c), creates the scalar stack variable whose address is held in \( %var1 \). Then, the value of \( %var \) is copied to \( %var1 \) (at the location of the \( \phi \) instruction) and \( %v1 \) is replaced with loads of \( %var1 \). Obviously, this increases the space consumed on the stack and the number of copy instructions. Hence, Sreedhar et al. [44] proposed using one of three modular copy placement algorithms and an SSA-based coalescing method, in order to reduce the number of copy instructions. Rastello [39] considered an SSA form constructed in machine-language IR and proposed an out-of-SSA translation that takes register constraints into account. To adapt out-of-SSA translation to just-in-time (JIT) compilation, various algorithms have been proposed to reduce the translation time [9,12].
Chapter 5. Out-of-ISSA Translation

(a) Program in SSA form.

(b) Program in Figure 5.1(a) after applying the out-of-SSA translation algorithm proposed by Cytron et al. [18].

(c) Program in Figure 5.1(a) after applying the out-of-SSA translation algorithm proposed by Sreedhar et al. [44].

Figure 5.1: Example illustrating translation out of SSA form.
Note that the additional store and load instructions can increase the size of the IR and reduce performance. These problems can be mitigated by coalescing variables during out-of-SSA translation [9, 39, 44] and coalescing registers during register allocation.

### 5.2.2 Challenges and Opportunities of Out-of-ISSA Translation

While previous work examined the use of ISSA for various analyses and optimizations [13, 33, 45], to the best of our knowledge, an out-of-ISSA translation algorithm was not reported. Liao [33] and Staiger [45] circumvented this problem by constructing ISSA in a separate data structure. Our initial out-of-ISSA translation algorithm extended the out-of-SSA translation algorithm by using scalar globals instead of scalar locals to propagate values across procedures.

Unfortunately, the resulting code was 1.5 times slower than the baseline, since translation out of ISSA form is more complex and poses additional problems. First, we must replace interprocedural references with variable accesses. The choice of variables impacts both the number and placement of copy instructions as well as the effectiveness of the compiler backend. Second, a naive replacement of the ISSA IR with equivalent instructions can significantly degrade performance. For instance, a drastic increase in copy instructions would be observed if we would simply replace each merge instruction with a new scalar global variable. Third, we cannot rely on the compiler backend to schedule newly inserted instructions or coalesce variables. For instance, the register allocator would coalesce variables mapped to registers, which does not include a global variable defined in one procedure and used in others. Moreover, a significant increase in the number of $\phi$ instructions can reduce the effectiveness of the register coalescer [34].

One way to resolve this problem is by updating code generation passes to work on ISSA IR, but this would involve substantial changes. In order to integrate ISSA into compilers and obtain performance improvement, these problems must be addressed.
While out-of-ISSA translation poses a number of challenges, it also presents a number of optimization opportunities. First, we can selectively introduce the store instructions that are required to pass values. Second, we can replace parameters with globals and vice versa. By exploiting these opportunities, we can reduce the number of parameters as well as store and load instructions.

5.2.3 Running Example

In order to illustrate the out-of-ISSA translation algorithm, we will use the example shown in Figure 5.2. In the C source code presented in Figure 5.2(a) the elements of structure $A$, which is allocated on the stack in procedure $main$, are initialized in procedure $init$ (by reading from a file stream) via calls to $getI$. Next, we call procedure $getCoefs$ and pass it structure $A$. In procedure $getCoefs$, $A.count$ coefficients are obtained via calls to procedure $getI$, scaled, and then assigned to the passed array. Note that a structure $St$ with size $sz$ and alignment $o$ is passed by value using $n = \frac{sz}{o}$ parameters. In order to do this, three actions are taken:

- The structure parameter in a procedure $P$ is replaced with $n$ integer (of size $o$) parameters $p_1, \ldots, p_n$.

- At call sites targeting $P$, the passed structure is cast into an integer (of size $o$) array with $n$ elements that are passed as arguments.

- In $P$, we allocate the structure $St$ on the stack. To initialize $St$, we cast $St$ to an integer (of size $o$) array and store $p_1, \ldots, p_n$ at their corresponding index.

The ISSA form for the C source code in Figure 5.2(a) is presented in Figure 5.2(b). The program starts by calling procedure $init$, where the record elements $count$, $num$, and $den$ of structure $A$ are defined and then propagated to usage points throughout the program. Once procedure $init$ returns, the stack variables $arr1$ and $arr2$ are assigned the
Chapter 5. Out-of-ISSA Translation

(a) C source code example.

```c
struct St { int count, num, den; }
void init(struct St* A) {
    A->count = getI();
    A->num = getI();
    A->den = getI();
}
void getCoefs(struct St A, int l, int h, int* arr) {
    for (int i = 0; i < A.count; ++i) {
        int num = getI()*(h-l);
        arr[i] = num*A.num/A.den;
    }
}
void main() {
    struct St A;
    init(&A);
    getCoefs(A,0,10,arr1);
    getCoefs(A,0,18,arr2);
    ...
    free(arr1);
    free(arr2);
}
```

(b) ISSA form for Figure 5.2(a).

```c
void init(struct St* %A) {
    %v0 := call @getI;
    %v1 := call @getI;
    %v2 := call @getI;
}
void getCoefs(int %a10, int %a11, int %a12, int %l, int %h, int* %arr) {
    %v3 := φ^V(CI2,#10),(CI3,#18);
    %v4 := φ^V(CI2,%v16),(CI3,%v17);
    %vA1 := sallocate #12;
    br BB2;
    BB0:
    %v5 := call @getI;
    %v6 := mul %v5,%v3;
    %v7 := mul %v6,%v1;
    %v8 := div %v7,%v2;
    %v9 := elemOf %v4,%v11;
    store %v9,%v8;
    %v10 := add %v11,#1;
    br BB2;
    BB1:
    %v11 := φ^V(BB0,#0), (BB1,%v10);
    %v12 := lt %v11,%v0;
    br %v12, BB1, . . . ;
    BB2:
    %v13 := sallocate #12;
    CI1: call @init, %vA;
    %v15 := mul %v5,%v3;
    %v16 := call @malloc, %v15;
    %v17 := call @malloc, %v15;
    CI2: call @getCoefs, %v0, %v1, %v2,
    CI3: call @getCoefs, %v0, %v1, %v2,
    CI4: . . .
    call @free, %v16;
    call @free, %v17;
}
```

Figure 5.2: Example to illustrate ISSA form and out-of-ISSA translation.
addresses of heap-allocated arrays. Since \texttt{arr1} and \texttt{arr2} are stack variables, their uses are replaced with their corresponding definitions during the construction of SSA form. Hence, the temporaries \%v16 and \%v17 on lines 32–33 in Figure 5.2(b) are inserted prior to constructing ISSA. In order to assign coefficients to the arrays, two calls to procedure \texttt{getCoefs} are made on line 34 and line 36 in Figure 5.2(b). Note that three parameters \%a10, \%a11, and \%a12 are used to propagate the value of the fields \textit{count}, \textit{num}, and \textit{den}, respectively. During ISSA construction, $\phi^V$ instructions were inserted to merge the values of these parameters, since procedure \texttt{getCoefs} is called twice. These $\phi^V$ instructions are folded and replaced during copy propagation, because \%a10, \%a11, and \%a12 are equal to \%v0, \%v1, and \%v2, respectively. Beyond this, procedure \texttt{getCoefs} has the $\phi^V$ defining \%v3 and \%v4, which merge the values of the parameters \%h and \%arr from the call sites \textit{CI}2 and \textit{CI}3.

\section{5.3 Applied Compiler Passes}

This section describes the passes we apply before and after constructing ISSA form as well as passes applied after the out-of-ISSA translation. In Section 5.3.1, we describe the storage-remap pass that converts stack and heap allocated SSA variables to globals. Section 5.3.2 describes passes applied on the IR in ISSA form and passes that are applied after the out-of-ISSA translation.

\subsection{5.3.1 Storage-Remap Transformation}

During the construction of ISSA form, the storage-remap pass replaces all newly handled SSA variables allocated on the stack or heap with globals. For each stack allocated SSA variable \textit{var} (that must be in a non-recursive procedure and whose address is taken or is a structure field) we allocate a global variable \textit{gv} of the same type, replace all uses of \textit{var} with \textit{gv}, and erase \textit{var} from its parent procedure. For each heap allocated SSA
struct St A, A1;
void init(struct St* %A) {
  %v0 := call @getI;
  S0: store* @A.count, %v0;
  %v1 := call @getI;
  S1: store* @A.num, %v1;
  %v2 := call @getI;
  S2: store* @A.den, %v2;
}

void getCoefs(int %a10, int %a11, int %a12, int %l, int %h, int* %arr) {
  BB0:
    %v3 := φ (CI2,#10), (CI3,#18);
    %v4 := φ (CI2, %v16), (CI3,%v17);
    S3: store* @A1.count, %v0;
    S4: store* @A1.num, %v1;
    S5: store* @A1.den, %v2;
    br BB2;
  BB1:
    %v5 := call @getI;
    %v6 := mul %v5, %v3;
    %v7 := mul %v6, %v1;
    %v8 := div %v7, %v2;
    %v9 := elemOf %v4, %v11;
    store %v9, %v8;
    %v10 := add %v11, #1;
    br BB2;
}

void main() {
  CI1: call @init, @A;
    %v15 := mul #4, %v0;
    %v16 := call @malloc, %v15;
    %v17 := call @malloc, %v15;
  CI2: call @getCoefs, %v0, %v1,
      %v2, #0, #10, %v16;
  CI3: call @getCoefs, %v0, %v1,
      %v2, #0, #18, %v17;
  ... call @free, %v16;
  call @free, %v17;
}

(c) ISSA form shown in Figure 5.2(b), when applying the storage-remap transformation and including SSA assignments (at labels S0,. . . ,S5).

Figure 5.2: Example to illustrate ISSA form and out-of-ISSA translation.

struct St A;
void init() {
  %v0 := call @getI;
  store @A.count, %v0;
  %v1 := call @getI;
  store @A.num, %v1;
  %v2 := call @getI;
  store @A.den, %v2;
}

void getCoefs(int %h, int* %arr) {
  BB0:
    %v3 := load @A.count;
    %v4 := load @A.num;
    %v5 := load @A.den;
    br BB2;
  BB1:
    %v6 := call @getI;
    %v7 := mul %v6, %h;
    %v8 := mul %v7, %v4;
    %v9 := div %v8, %v5;
    %v10 := elemOf %arr, %v12;
    store %v10, %v9;
    %v11 := add %v12, #1;
    br BB2;
  BB2:
    %v12 := φ (BB0, #0), (BB1, %v10);
    %v13 := lt %v12, %v3;
    br %v13, BB1, . . . ;
}

int main() {
  call @init;
  %v14 := load @A.count;
  %v15 := mul #4, %v14;
  %v16 := call @malloc, %v15;
  %v17 := call @malloc, %v15;
  call @getCoefs, #10, %v16;
  call @getCoefs, #18, %v17;
  ... call @free, %v16;
  call @free, %v17;
}

(d) Resulting SSA form after out-of-ISSA translation.
variable \( \text{var} \), we first determine its type, \( T_y \), by using the point-to graph to identify all casts of \( \text{var} \) and selecting the type with the largest size. The type \( T_y \) is padded with a character array (at the end) to match the size of \( \text{var} \). Then a global variable \( \text{gv} \) with type \( T_y \) is created and used to replace all uses of \( \text{var} \) with bit-casted versions of \( \text{gv} \). Lastly, since \( \text{gv} \) is a global variable it cannot be deallocated; this poses a problem, as the memory space for \( \text{var} \) is eventually deallocated. To prevent the deallocation of \( \text{gv} \), we guard against the invocation of memory deallocation routines where the argument being freed, \( \text{arg} \), points-to \( \text{var} \). When \( \text{arg} \) can only point-to \( \text{var} \), the deallocation call is removed. Otherwise, we predicate the deallocation call so that it is only executed when \( \text{arg} \neq \text{gv} \).

This transformation simplifies other passes as the address of every SSA variable is a unique constant. Furthermore, we reduce the number of instructions in the program and obviate the need to propagate the address of SSA variables, via other variables, parameters, and return values. This allows us to eliminate a number of arguments, storage instructions, and load instructions. We suspect that this transformation also reduces register pressure and spilling, because a large number of pointer values are folded to constants.

**Example 5.1 Storage-remap transformation on program in Figure 5.2**

In Figure 5.2(a), two structures are allocated on the stack on line 7 in procedure getCoefs and line 16 in procedure main. As shown in Figure 5.2(b), if we would construct ISSA without the storage-remap transformation two stack allocation instructions would be inserted on lines 29 and 12 (and the addresses are assigned to the temporaries \( \%vA \) and \( \%vA1 \)).

Since procedures getCoefs and main are not recursive, these structures are converted to global variables. Therefore, in Figure 5.2(c), the storage-remap transformation replaces \( \%vA \) and \( \%vA1 \) with pointers to the global variables \( A \) and \( A1 \), respectively.
5.3.2 Applied Passes

Prior to constructing ISSA form, we apply standard intraprocedural passes followed by the storage-remap pass. After ISSA construction, we apply copy propagation, the one-level context-sensitive constant propagation, global common subexpression elimination (GCSE), and dead code removal (ADCE pass).

This is followed by performing the out-of-ISSA translation that converts the IR back to SSA form. After the out-of-ISSA translation a large number of instructions, parameters, and return values become redundant. We apply the dead argument elimination pass and the ADCE pass to remove redundant parameters, arguments, return values, various instructions, and their operands. Lastly, we remove program variables that are not used.

5.4 Proposed Algorithm

5.4.1 Overview

An out-of-ISSA translation must remove uses of temporaries holding the values of $\phi^S$, $\phi^L$, $\phi^V$, and $\phi^C$ instructions as well as interprocedural references. This is done by replacing uses of these temporaries with program variable accesses. In particular, we use parameters, return values, and SSA variables to replace references to such temporaries. These program variables are referred to as propagation variables in this chapter.

Let us assume that during ISSA form construction we did not remove any of the original store instructions, parameters, and return values. In such a scenario, each SSA variable would contain its value at any of its original usage program points. We can translate such an ISSA form program into SSA form in the following manner:

- Replace uses of temporaries that are defined in other procedures (interprocedural references) with the propagation variable that is equal to them at the given usage point.
• Replace each use of every temporary holding the value of a $\phi^V$ or $\phi^C$ instruction $I$ with an access to the SSA variable for which $I$ was inserted.

• Replace each use of every temporary holding the value of a $\phi$ instruction $I$ that is inserted during ISSA form construction with an access to the SSA variable for which $I$ was inserted.

• Let us assume $%I0 = \phi^S pv', @var, val, curr$. Uses of $%I0$ are replaced with accesses to $var$ and the store instruction ($store pv, val$) that corresponds to the $\phi^S$ instruction is retained.

• Let us assume $%I0 = \phi^L pv, \langle @var_1, val_1 \rangle, \ldots, \langle @var_n, val_n \rangle$. We replaced $%I0$ with a load instruction whose pointer value is $pv$.

Note that when constructing ISSA form, references to parameters and return values are replaced by inserting $\phi^V$ and $\phi^C$ instructions. Moreover, we also remove store instructions whose pointer value is the address of an SSA variable. In order to translate out-of-ISSA in the aforementioned manner, we have to make various changes to the ISSA construction algorithm, keep track of certain information, and leverage new analyses that are outlined below:

• During ISSA construction, we have to keep track of all the original store instructions, parameters, and return values.

• Determine or keep track of the SSA variable for which each $\phi^V$, $\phi^C$, and $\phi$ instructions is inserted for.

• We need an analysis to determine the propagation variables that contain the value of a temporary at a given program point.

• A mechanism to ensure that an SSA variable $var$ contains its value at a given program point $PP$. Note that this condition ($var$ contains its value at $PP$) is always
satisfied if we do not remove any of the original store instructions, parameters, and return values. However, if we choose to selectively introduce the original store instructions (as we do in the proposed algorithm) and remove redundant parameters and return values, then a mechanism to ensure that \( \text{var} \) contains its value at \( PP \) is necessary.

- Set of heuristics and analyses to judiciously choose the propagation variable that replaces a given interprocedural reference.

At a high level, our out-of-ISSA translation algorithm takes this approach while trying to minimize the number of store instructions, parameters, and return values in the resulting code. In order to address the first two challenges we keep track of various information during ISSA form construction as described in Section 5.4.2. Note that the ISSA form construction introduces store and load instructions around call instructions invoking library procedures. In Section 5.4.3, we explain how these store and load instructions are removed during the out-of-ISSA translation. A high-level flow diagram illustrating the proposed out-of-ISSA translation algorithm can be found in Figure 5.3.

In the first step, we judiciously choose the propagation variable and make certain that it contains the value of the temporary it replaces by introducing store instructions. To replace a temporary \( \%I0 \), the value map is leveraged to identify the set of propagation...
variables that may be equal to \( I_0 \). The incoming map is used to identify the value of a propagation variable at a certain program point \( PP \) (to narrow down the choice of propagation variables) and the store instructions that need to be introduced in order to propagate its value to \( PP \). At the end, we output an IR in ISSA form that includes store instructions to SSA variables. Moreover, the output also consists of \( VS \) (variable selector), which maps uses of temporaries to the propagation variable chosen to replace them. Once the first step is completed, we use \( VS \) to replace the uses of temporaries with the variable to which they are mapped.

In the rest of this section, we present our algorithm in detail. In Section 5.4.4, we establish a framework to translate out of ISSA. In Section 5.4.5, we describe how we choose the propagation variable. In Section 5.4.6, we present the algorithm for introducing store instructions. In Section 5.4.7, we outline our approach to removing ISSA IR extensions and replacing interprocedural references. In Section 5.4.8, we discuss the impact of our passes on the IR in Figure 5.2(c).

### 5.4.2 Simplifications

The placement of store instructions and variable coalescing impacts the performance of the program. In order to obtain better program performance, we may have to explore multiple solutions using program analyses that are computationally expensive. As such, we simplify the out-of-ISSA translation in two ways:

1. We keep track of the original store instructions (prior to ISSA construction) whose pointer value is an SSA variable. For a given store instruction assigning \( val \) to SSA variable \( var \) at program point \( PP \) we introduce the SSA assignment \( store* var, val \) at \( PP \). The SSA assignment is an instruction that does not have any effect on the state of the program and does not have a value associated with it (i.e. held in a temporary). In Figure 5.2(c), the instructions at program points \( S_0-S_5 \) (on lines 4–8 and lines 14–16) are SSA assignments.
2. We keep track of the propagation variable that corresponds to $\phi$, $\phi^V$, and $\phi^C$ instructions using the mapping $PhiVar$. In Figure 5.2(c), the $\phi^V$ instructions assigned to $%v3$ and $%v4$ correspond to the parameters $%h$ and $%arr$, respectively. Hence, $PhiVar[%v3] = %h$ and $PhiVar[%v4] = %arr$.

With these simplifications, we do not have to coalesce variables or determine the placement of store instructions. This will enable us to convert the IR in ISSA form to an SSA form that at least matches the performance of the original IR (prior to ISSA construction).

### 5.4.3 Library Calls

During the dereference conversion step (Section 4.4) of ISSA form construction we insert load and store instructions whose pointer value is the address of SSA variables.

Store instructions are inserted in order to write the value of SSA variables that may be used by the library procedure that is invoked. Let us consider one such store instruction at program point $PP$ which writes the value $val$ to an SSA variable $var$. During the out-of-ISSA translation we treat this store instruction as a use of $var$, hence, the out-of-ISSA translation ensures (by converting SSA assignments to store instructions) that $var$ is equal to its value ($val$) at $PP$. By doing this, these store instructions become redundant and are removed.

Load instructions are inserted after the library call in order to retrieve the value of SSA variables that may be defined by the library procedure that is invoked. Lets us consider such a load instruction, that defines the temporary $%I0$ and whose pointer value is the address of the SSA variable $var$. We treat this load instructions as a definition of $var$, thus enabling us to replace interprocedural references to $%I0$ using accesses to $var$.  

5.4.4 Framework

This chapter uses various concepts introduced in Chapter 2, in particular sets and procedures described in Section 2.3. In addition, we establish the following framework to present the proposed algorithm:

- **Vars** is the set of SSA variables.
- **Params** is the set of parameters. To simplify the presentation, we refer to a parameter using the temporary that holds its value.
- **RetVals** is the set of return values. To simplify the presentation, we refer to a return value from a call instruction $ci$ using the temporary holding the value of $ci$.
- $\mathcal{PV} = \{\text{Vars} \cup \text{Params} \cup \text{RetVals}\}$ is the set of propagation variables.
- $\text{getCorrespondingVar} : \mathcal{TMP} \mapsto \mathcal{PV}$ is a function that returns the propagation variable that a temporary $%I0 \in \mathcal{TMP}$ corresponds to. If $%I0$ holds the value of the instruction $\phi^S \text{pv, var, ...}$, then var is returned. Otherwise, $\text{PhiVar}[%I0]$ is returned.

Value Map

Using Algorithm 5.1 we derive the value map $\mathcal{VM} : \mathcal{TMP} \mapsto \text{powerset}(\mathcal{PV})$, which is a mapping between a temporary $%I0 \in \mathcal{TMP}$ used in other procedures and the propagation variables that may be equal to it at some program point. The value map is constructed during an IR traversal. If a parameter or return value $\text{var}$ passes the temporary $%I0$ (i.e. corresponding $\phi^V$ or $\phi^C$ instruction folded to $%I0$), then we insert $\text{var}$ into $\mathcal{VM}[%I0]$. If an SSA assignment stores a temporary $%I0$ into $\text{var}$, then we insert $\text{var}$ into $\mathcal{VM}[%I0]$. 
Algorithm 5.1 Deriving the value map.

Output: \( VM : \mathcal{TM} \mapsto \text{powerset}(\mathcal{PV}) \)

1: \quad \text{foreach procedure } P \text{ in the program do}
2: \quad \text{foreach parameter } %par \text{ of } P \text{ do}
3: \quad \quad \text{if each argument passed through } %par \text{ is equal to a temporary } %I0 \text{ then}
4: \quad \quad \quad VM[%I0] := VM[%I0] \cup %par
5: \quad \text{foreach instruction } I \text{ in procedure } P \text{ do}
6: \quad \quad %I0 := \text{InstToTemp}(I)
7: \quad \quad \text{if } I \text{ is a call instruction then}
8: \quad \quad \quad \text{if each procedure called returns the same temporary } %J0 \text{ then}
9: \quad \quad \quad \quad VM[%J0] := VM[%J0] \cup %I0
10: \quad \quad \quad \text{else if } I \text{ is the SSA assignment store* } var, %I0 \text{ then}
11: \quad \quad \quad \quad VM[%I0] := VM[%I0] \cup var

Example 5.2 The value map derived when iterating over the IR in Figure 5.2(c).

\[
VM[\%v0] = \{@A.count, @A1.count, \%a10\}
\]
\[
VM[\%v1] = \{@A.num, @A1.num, \%a11\}
\]
\[
VM[\%v2] = \{@A.den, @A1.den, \%a12\}
\]

Note that we maintain the addresses of SSA variables. For instance, @A.count, @A.num, and @A.den are the addresses (constants) of the fields count, num, and den within structure A. Moreover, @A1.count, @A1.num, and @A1.den are the addresses of the fields count, num, and den within structure A1.

Incoming Map

In ISSA, each use of an SSA variable is replaced with a single definition. The incoming map enables us to derive the value \( val \) of an SSA variable \( var \in Vars \) at a program point \( PP \in \mathcal{L} \) as well as the store instructions that have to be introduced in order make sure that \( var \) is equal to \( val \) at \( PP \).

The incoming map is \( IM : \mathcal{L} \times Vars \mapsto \text{INST} \). At call sites and procedure entries \( (PP \in \mathcal{L}) \), we maintain a mapping between each SSA variable \( var \in Vars \) and its
definition at $PP$. Originally, an SSA variable may have multiple reaching definitions at a program point. However, during the construction of ISSA $\phi$, $\phi^S$, $\phi^V$, and $\phi^C$ instructions are inserted at merge points so that each use of $var$ is replaced with a temporary defined once.

Conceptually, the incoming map represents all the reaching definitions of $var$ at $PP$ using a single instruction. Note that an SSA variable $var$ is originally assigned at SSA assignments and $\phi^S$ instructions. If $var$ has a single reaching definition at $PP$ then it will be mapped to an SSA assignment. When $var$ has multiple reaching definitions at $PP$, it will be mapped to a $\phi^S$, $\phi$, $\phi^V$, or $\phi^C$ instruction.

The incoming map is equivalent to a reaching definition analysis, with the one exception that we represent multiple reaching definitions with a single instruction. Since $\phi$, $\phi^V$, or $\phi^C$ instructions can be folded, multiple SSA assignments that assign the same value $V$ to a given variable $var \in \textit{Vars}$ can reach a program point. In order to handle this case, we introduce the quasi $\phi$ instruction, which merges the same value and becomes the single definition of $var$ at $PP$.

The incoming map is computed by applying two passes. A bottom-up pass is first applied over the acyclic call graph to determine the definition of SSA variables at the end of each procedure $P$; and passes it to call sites targeting $P$. Then, a top-down pass is applied over the acyclic call graph to propagate the definitions of each SSA variable down both the call graph and control flow graph.

Note that parameters and return values are directly associated with their corresponding procedure and call site, respectively. Hence, to improve efficiency, we omitted them from the incoming map.

**Example 5.3** The incoming map for the IR in Figure 5.2(c).

When examining the IR in Figure 5.2(c), we create a number of entries in $IM$, however the only relevant program points are after the call site $CI1$ (on line 33 in Figure 5.2(c)) and the entry to procedure getCoefs where:
\[ IM[(CI1, \@A.count)] = IM[(getCoeFs, \@A.count)] = \text{ProgPointToInst}(S0) \]
\[ IM[(CI1, \@A.num)] = IM[(getCoeFs, \@A.num)] = \text{ProgPointToInst}(S1) \]
\[ IM[(CI1, \@A.den)] = IM[(getCoeFs, \@A.den)] = \text{ProgPointToInst}(S2) \]

5.4.5 Selecting the Propagation Variable

Algorithm 5.2 selects the propagation variable which will be used to remove ISSA IR as well as interprocedural references. Algorithm 5.2 is an iterative worklist algorithm that accepts as input ISSA IR as well as the value map, incoming map, and the interprocedural value replacement map (IVR). It judiciously chooses the propagation variable and introduces the required store instructions.

Prior to selecting propagation variables, we fold \( \phi^C \) instructions by leveraging the IVR data structure in Algorithm 5.2, line 1. Let us assume that \( I_0 \) holds the value of a \( \phi^C \) instruction merging a single value \( V \) while the SSA assignment store* \( \text{var,} \ I_0 \) is located at program point PP. By applying this step we simplify the propagation variable selection, since we can be certain that \( \text{var} \) cannot propagate the instance of \( V \) that \( I_0 \) is equal to at program points following PP.

At first, on lines 3–12 in Algorithm 5.2, we replace intraprocedural uses of temporaries assigned ISSA instructions. When a temporary \( I_0 \) holds the value of a \( \phi, \phi^V, \phi^C, \text{or} \phi^S \) instruction \( I \), then we identify the corresponding propagation variable of \( I_0 \) \( (\text{var}) \) and commit it. Committing the value \( \text{val} \) in (a propagation variable) \( \text{propvar} \) at a program point PP, will ensure that \( \text{propvar} \) is equal to \( \text{val} \) at PP. To commit a propagation variable, we identify required parameters and return values (by adding interprocedural references to UsefulRefs) and convert needed SSA assignments to store instructions. Committing \( I_0 \) in \( \text{var} \) at InstToProgPoint(\( I \)) will enable us to replace \( I_0 \) with an access to \( \text{var} \) within the parent procedure of \( I \) and as such, we map each intraprocedural...
Algorithm 5.2 Propagation variable selection.

Input: $VM$, $IM$, $IVR$

Output: $VS : I\mathcal{NS}\mathcal{T} \times \mathcal{T}MP \mapsto PV$

1. Use $IVR$ to fold every possible $\phi_C$ instructions
2. $VS := $UsefulRefs $:= \emptyset$
3. foreach procedure $P$ do
4.   foreach instruction $I$ in procedure $P$ do
5.     if $I$ is a $\phi^S$, $\phi^V$, $\phi^C$, or $\phi$ instruction then
6.       $%I0 = $InstToTemp($I$)
7.       if ($var := $getCorrespondingVar($%I0$)) $\neq \emptyset$ then
8.         CommitVar($I$, $%I0$, $var$, $UsefulRefs$)
9.     foreach instruction $U$ in procedure $P$ that uses $%I0$ do
10.    $VS[[U,$%I0$]] := var$
11.   else if $I = \phi^L \ldots$ then
12.     CommitVarRecur($I$, $InstToTemp(I)$, $\ldots$)
13. Changed $:= true$
14. while Changed do
15.   Changed $:= false$
16. foreach Changed instruction $I$ inside procedure $P$ do
17.   foreach Interprocedural reference between instruction $I$ and a temporary $Op$ do
18.     if isUsefulReference($I, Op$) then
19.       $PP := $getPropagationPoint($I, Op$)
20.       $PossibleVars := $getPossibleVars($Op$)
21.       $PropVars := $getPropVars($PossibleVars, Op, PP$)
22.       $var :=$judiciouslyChoose($PropVars, Op$)
23.       $VS[[I,$Op$]] := var$
24.     if CommitVar($I$, $Op$, $var$, $UsefulRefs$) then
25.       Changed $:= true$
references \( \langle U, \%I0 \rangle \) to \( \text{var} \) in \( \mathcal{VS} \). If \( I \) is a \( \phi^L \) instruction, then we commit each of its possible values to their associated SSA variable. This will enable us to replace the \( \phi^L \) instruction with a load of its pointer value.

Afterwards, we apply an IR traversal that selects the variables that are used to replace interprocedural references. In Algorithm 5.2, lines 18–25, we limit our focus to instruction \( I \) in procedure \( P \) and one of its operands, a temporary \( Op \), which is defined in procedure \( Q \neq P \). During our algorithm, we commit propagation variables, which may require us to replace additional interprocedural references. As such, our algorithm iterates over the IR until no additional interprocedural references are localized. In the first iteration of the loop on lines 16–25 we visit all the instructions. To improve the efficiency of the algorithm, we keep track of the changed instructions and newly introduced instructions and in successive iterations of the loop we visit just these instructions. Note that not all interprocedural references are replaced. In particular, on line 18 we call the procedure \( \text{isUsefulReference} \), which returns false when:

- The temporary assigned the result of computing \( I (\%I0 = \text{InstToTemp}(I)) \) corresponds to a propagation variable \( \text{var} \). In this case, intraprocedural references to \( \%I0 \) are mapped to \( \text{var} \) while interprocedural references to \( \%I0 \) will be replaced. Therefore, \( \%I0 \) will not be used in the program and as such, the interprocedural reference to \( Op \) does not have to be replaced.

- \( I \) is a \( \phi^L \) instruction and \( Op \) is not the pointer value. In this case, \( Op \) can be ignored, since \( \%I0 = \text{InstToTemp}(I) \) is replaced with a load of the pointer value.

- \( I \) is a call instruction and \( Op \) is an argument that is not used. This condition is tested by leveraging the set \( \text{UsefulRefs} \).

- \( I \) is a return instruction and none of the call instructions that target \( P \) use its return value. This condition is tested by leveraging the set \( \text{UsefulRefs} \).
• \( I \) is an SSA assignment.

Once we determine that the interprocedural reference from \( I \) to \( Op \) is useful, we apply the steps on lines 19–25. Below, in sequence, we elaborate on each step. Example 5.4 will then illustrate how these steps are applied to the IR in Figure 5.2(c).

**Computing the propagation point**

On line 19 in Algorithm 5.2, we compute \( PP \), the propagation point of \( Op \) to instruction \( I \) using procedure \( \text{getPropagationPoint} \), which is presented in Algorithm 5.3. First, we check whether \( P \) and \( Q \) are in the same call graph SCC. If this is the case, then \( Op \) can only be passed to \( I \) through the entry of procedure \( P \), because a \( \phi^C \) instruction whose parent procedure is \( P \) cannot be replaced with a value defined in a SCC to which \( P \) belongs. Otherwise, let us assume that \( \text{ReachProcs}(P, Q) \) is the set of call instructions in procedure \( P \) that can reach procedure \( Q \). If \( \text{ReachProcs}(P, Q) = \emptyset \), then the propagation point is the entry of procedure \( P \) because none of the call instructions in \( P \) can reach procedure \( Q \). In this case, the loop on line 6 in Algorithm 5.3 will never execute and as such, no assignments to \( \text{FirstDom} \) are made on line 8. Therefore, the algorithm returns the entry to procedure \( P \) on line 9. Otherwise, if \( \text{ ReachProcs}(P, Q) = \{ci_1, \ldots, ci_n\} \), then the propagation point of \( Op \) is at a call instruction \( ci_k \), where \( 1 \leq k \leq n \). Let us assume that a \( \phi^C \) instruction at the call site of \( ci_k \) was replaced with a value defined in \( Q \) at instruction \( I \). In this case, \( ci_k \) must dominate \( I \) and none of the call instructions in \( \text{ReachProcs}(P, Q) - ci_k \) can be on any path between \( ci_k \) and \( I \). Therefore, we identify the propagation point by checking that two conditions are satisfied:

1. \( ci_k \) dominates \( I \).

2. \( ci_k \) does not dominate any (other) call site \( ci_j \neq ci_k \) that also dominates \( I \).
Algorithm 5.3 Computing the propagation point of a temporary Op that is defined in procedure Q and used at instruction I in procedure $P \neq Q$.

1: proc getPropagationPoint(I : INST, Op : TEMP) : L begin
2: FirstDom := entry(P)
3: if P and Q are in a call graph SCC then
4: return InstToProgPoint(entry(P))
6: foreach call instruction ci, where $Q \in RPC[ci]$ do
7: if ci = UsagePP and ci dominates UsagePP and FirstDom dominates ci then
8: FirstDom := ci
9: return InstToProgPoint(FirstDom)
10: end

Computing the Set Of Propagation Variables Holding Op

On line 20 in Algorithm 5.2, we compute PossibleVars, which is the set of variables Op is assigned to. At first, PossibleVars is assigned $VM[Op]$. Then, if TempToInst(Op) is a $\phi^C$ instruction that merges a single value val, then val can also be propagated using variables holding val. Note that val must be a temporary because we folded all $\phi^C$ instructions, and as such we add $VM[val]$ to PossibleVars.

In addition, we test whether getCorrespondingVar(Op) maps to a propagation variable var. In this case, var will be equal to Op if we commit Op to var. Hence, we add var to PossibleVars.

Computing the Set Of Propagation Variables Holding Op at the Propagation Point

On line 21 in Algorithm 5.2, we derive PropVars $\subseteq$ PossibleVars, which is the subset of PossibleVars that hold Op at the propagation point PP. Algorithm 5.4 presents procedure getPropVars, which is used to derive PropVars from PossibleVars.

A parameter $\%param \in$ PossibleVars is added to PropVars, if $\%param$ is a parameter in procedure $P$ and PP is the entry to $P$. A return value $\%ci \in$ PossibleVars at call site $cs$ can be added to PropVars if PP is equal to $cs$. Finally, if propvar $\in$ PossibleVars is an SSA variable, we check its definition $DI = IM[\langle PP, var \rangle]$. We can add propvar
Algorithm 5.4 Deriving propagation variables. Procedure isSameValue returns true if the value of the passed temporary is identical at both program points and false otherwise.

1: proc getPropVars(PossibleVars : powerset($\mathcal{P}V$),
            $Op : T,MP,PP : L$) : powerset($\mathcal{P}V$) begin
2:   PropVars := $\emptyset$
3:   foreach propvar $\in$ PossibleVars do
4:     if propvar is a parameter in procedure $P$ and $PP$ is the entry to $P$ then
5:       PropVars := PropVars $\cup$ propvar
6:     else if TempToInst(propvar) = call... ∧
7:       InstToProgPoint(TempToInst(propvar)) = $PP$ then
8:       PropVars := PropVars $\cup$ propvar
9:     else if propvar is an SSA variable then
10:    DI := $\mathcal{T}M[\langle PP, propvar \rangle]$
11:    if InstToTemp(DI) = $Op$ then
12:      PropVars := PropVars $\cup$ propvar
13:    else if isSameValue($\mathcal{IVR}, Op, InstToProgPoint(DI), PP$) then
14:      if DI is a quasi $\phi$ instruction merging $Op$ or $DI$ = store$^*$ propvar, $Op$ then
15:        PropVars := PropVars $\cup$ propvar
16:      else if $Op = \phi^C(\ldots, V) \land DI =$ store$^*$ propvar, $V$ then
17:        if isSameValue($\mathcal{IVR}, V, InstToProgPoint(TempToInst(Op)),
                        InstToProgPoint(DI)) then
18:          PropVars := PropVars $\cup$ propvar
19:   end
20: return PropVars
21: end
to $PropVars$, if:

- $Op$ is the temporary holding the value of $DI$.
- $DI$ is a quasi $\phi$ instruction that merges the single operand $Op$ and the value of $Op$ is the same at $PP$ and the program point of $DI$.
- $DI$ is an SSA assignment that stores $Op$ and the value of $Op$ is the same at both $PP$ and the program point of $DI$.
- $Op$ is a temporary assigned the result of a $\phi^C$ instruction that merges (a single value) $V$, $DI$ is an SSA assignment that stores $V$, the value of $Op$ is the same at both $PP$ and the program point of $DI$, and the value of $V$ is the same at the program point of $DI$ and the program point where $Op$ is defined.

**Choosing the Propagation Variable**

On line 22 in Algorithm 5.2, we choose the replacement variable $var \in PropVars$. This is done by invoking the procedure $judiciouslyChoose$, which consists of a sequence of conditions that are tested in the order presented below. Once $var$ is selected, subsequent conditions are skipped.

1. If $PropVars$ contains a single variable, then we choose it.

2. If $Op$ holds the value of a $\phi^V$, $\phi^C$, $\phi$, or $\phi^S$ instruction, and $var \in PropVars$ is its corresponding propagation variable, then we choose $var$.

3. If $PropVars$ contains multiple SSA variables, then we try to choose the SSA variable whose definition is nearest to $Op$. Choosing the SSA variable with the nearest definition would usually result in fewer variables propagating $Op$ and hence, fewer load and store instructions. If $PropVars$ contains only one SSA variable $var$, then we choose $var$. 
In order to estimate the nearest definition, we use heuristics. First, if it exists, we try to find the nearest definition within procedure \( Q \) (the procedure in which \( Op \) is defined). If more than one definition is located in \( Q \), we use the topological order of basic blocks to estimate the definition that is nearest to the definition of \( Op \).

Otherwise, none of the definitions are located in procedure \( Q \). In this case, we approximate the order of the SSA variable definitions, by leveraging the incoming map. If an SSA variable \( var1 \) already contains \( Op \) at the definition of another SSA variable \( var2 \), then we presume that \( var1 \) is defined prior to \( var2 \). More formally, at the definition of each SSA variable \( var \in PropVars \), we test whether \( Op \) is not contained in any other SSA variable (i.e. \( PropVars - var \)) by querying the incoming map. We return the first SSA variable satisfying this condition (or a random one otherwise).

4. Lastly, we choose any parameter or return value available.

**Example 5.4 Choosing propagation variables in Figure 5.2(c)**

Note that \( \%v3 \) holds the value of a \( \phi^V \) instruction and \( \PhiVar[\%v3] = \%h \). As such, we choose the parameter \( \%h \) to replace \( \%v3 \) at the instruction \( \text{TempToInst}(\%v6) \) by setting \( VS[\text{TempToInst}(\%v6), \%v3] := \%h \) in Algorithm 5.2, line 3. For the same reason, we choose the parameter \( \%arr \) to replace \( v4 \) by setting \( VS[\text{TempToInst}(\%v9), \%v4] := \%arr \).

In Figure 5.2(c), let us examine the multiplication instruction defining \( \%v7 \) on line 21. This instruction has an interprocedural reference to its operand \( \%v1 \), because \( \%v1 \) is defined outside procedure getCoefs. In order to replace this interprocedural reference, we apply the following steps:

1. We compute the propagation point of \( \%v1 \) at its use on line 21, which is the entry to procedure getCoefs.

2. We compute \( possibleVars \) by querying \( VM \). When performing this query, we note
that the value of \( \%v1 \) is assigned to the SSA variables whose address is \(@A\text{.num}\) and \(@A1\text{.num}\) as well as the parameter \( \%a11 \).

3. We derive \( \text{PropVars} \subseteq \text{PossibleVars} \). First, since \( \mathcal{I}M[(\text{getCoefs}, @A1\text{.num})] = \emptyset \), we eliminate \(@A1\text{.num}\) from consideration. We include \(@A\text{.num}\) in \( \text{PropVars} \) because \( \mathcal{I}M[(\text{getCoefs}, @A\text{.num})] \) is equal to the SSA assignment \( \text{ProgPointToInst}(S1) \) and the value of \( \%v1 \) is the same at both \( S1 \) and the entry into procedure get-Coefs. We also add \( \%a11 \) to \( \text{PropVars} \), since the propagation point is the entry to getCoefs and \( \%a11 \) is a parameter in getCoefs.

4. We choose the SSA variable whose address is \( @A\text{.num} \) because SSA variables have a higher priority and \( S1 \) is located immediately after the definition of \( \%v1 \). This is done by setting \( VS[\text{TempToInst}(\%v7), \%v1] := @A\text{.num} \).

Using similar reasoning, we choose the propagation variable (whose address is) \( @A\text{.den} \) to replace the interprocedural reference between the division instruction on line 22 and the temporary \( \%v2 \).

5.4.6 Committing Propagation Variables

Algorithm 5.5 presents the procedure \( \text{CommitVar} \), which is used to commit the propagation variable. At first, we test whether \( pvar \) is a parameter or a return value and call procedure \( \text{CommitParamRet} \) on line 5, if this is true. This procedure ensures that the interprocedural reference (at call and return instructions) that is needed to propagate the parameter or return value is added to \( \text{UsefulRefs} \). Otherwise, \( pvar \) is an SSA variable and we must be certain that \( pvar \) is equal to \( Op \) at instruction \( I \). As previously mentioned, temporaries holding the value of \( \phi, \phi^S, \phi^V, \) and \( \phi^C \) instructions are replaced with their corresponding propagation variable. If \( Op \) holds the value of such instructions and \( var \) is its corresponding propagation variable, then we must ensure that \( \text{var} = Op \) at the program point where \( Op \) is defined (\( \text{ProgPointToInst(TempToInst(Op))} \)). Moreover,
Algorithm 5.5  Procedure CommitVar. It introduces store instructions and marks useful parameters and return values so that $pvar$ is equal to $Op$ at $\text{InstToProgPoint}(I)$.

1: $\text{HashCommitRec} := \emptyset$
2: proc CommitVar($I : \text{INST}, Op : \text{TMP}, pvar : \mathcal{PV}$, $\text{UsefulRefs} : \text{powerset}(\text{INST} \times \text{TMP})$) : bool begin
3:   $\text{Changed} := \text{false}$
4:   if $pvar \in \text{Params} \lor pvar \in \text{RetVals}$ then
5:     $\text{Changed} := \text{CommitParamRet}(pvar, \text{UsefulRefs})$
6:   else
7:     if $Op \in \text{TMP} \land \text{TempToInst}(Op) \notin \text{HashCommitRec}$ then
8:       $\text{HashCommitRec} := \text{HashCommitRec} \cup \text{TempToInst}(Op)$
9:     if CommitVarRecur($\text{TempToInst}(Op)$) then
10:        $\text{Changed} := \text{true}$
11:   if $Op \notin \text{TMP} \lor \text{TempToInst}(Op) \neq I$ then
12:      foreach SSA assignment $DI$ that assigns $Op$ to $pvar$ and reaches $I$ do
13:        convert $DI$ to a store instruction
14:        $\text{Changed} := \text{true}$
15:   return $\text{Changed}$
16: end

if $Op$ holds the value of a $\phi^L$ instruction, we must make sure that each aliased SSA variable contains its value at $\text{ProgPointToInst}($TempToInst$(Op))$. In Algorithm 5.5, this is done by calling procedure CommitVarRec on line 9, which may introduce multiple store instructions. Afterwards, we make sure that $pvar$ is equal to $Op$ at $I$ by converting SSA assignments (assigning $Op$ to $pvar$) that can reach $I$ to store instructions. On line 15, procedure CommitVar returns $\text{Changed}$, which is equal to true when new store instructions are introduced or interprocedural references are added to $\text{UsefulRefs}$ and to false, otherwise. This will ensure that Algorithm 5.2 selects a propagation variable for each useful interprocedural reference.

Procedure CommitParamRet is presented in Algorithm 5.6. On lines 3–7, we commit parameters. In order to commit a parameter $par$ whose parent procedure is $P$, we visit each call instruction $ci$ that can call $P$ on line 4. Let us assume that the parent procedure of $ci$ is $Q$ and the operand passed to $par$ is a temporary $\%arg$. If $\%arg$ is defined in a procedure $R \neq Q$, then we add $\langle ci, \%arg \rangle$ to $\text{UsefulRefs}$. Therefore, on line 18 in Algorithm 5.2, the procedure call to isUsefulReference will return true
Algorithm 5.6 Commit parameters and return values.

1: proc CommitParamRet(pvar : PV, UsefulRefs : powerset(INST × TMP)) : bool
2: begin
3: Changed := false
4: if pvar ∈ Params then
5: foreach call instruction ci that invokes parent(pvar) do
6: if ci has an interprocedural reference to %arg, the operand passed to pvar then
7: Changed := ⟨ci, %arg⟩ ∉ UsefulRefs
8: UsefulRefs := UsefulRefs ∪ ⟨ci, %arg⟩
9: else if pvar is a return value from call instruction ci then
10: foreach return instruction RI in a procedure called by ci do
11: if RI has an interprocedural reference to its %rval then
12: Changed := ⟨RI, %rval⟩ ∉ UsefulRefs
13: UsefulRefs := UsefulRefs ∪ ⟨RI, %rval⟩
14: return Changed
15: end

when passed ⟨ci, %arg⟩. In a similar manner, we commit return values by adding the interprocedural references between required return instructions and their operand to UsefulRefs. Procedure CommitParamRet will return true when one or more additional interprocedural references are added to UsefulRefs and false otherwise.

Example 5.5 Committing parameters in the ISSA form in Figure 5.2(c)

Recall that VS[TempToInst(%v6), %v3] = %h and VS[TempToInst(%v9), %v4] = %arr. The process of committing both parameters follows identical steps so we focus on the parameter %h.

For the parameter %h, procedure CommitVar is called in Algorithm 5.2, line 8 and passed TempToInst(%v3), %v3, %h, and UsefulRefs. Since pvar in procedure CommitVar (Algorithm 5.5) holds the value of a parameter (%h), procedure CommitParamRet is called on line 5. Because %h is a parameter in procedure getCoefs, on lines 3–7 in procedure CommitParamRet (Algorithm 5.6) we visit call instructions that invoke the procedure getCoefs. However, the values passed for %h at these call instructions is 10 and 18, which are constants and not interprocedural references. As such, no additional interprocedural references are added to UsefulRefs.
Algorithm 5.7 Procedure CommitVarRecur. It will enable us to replace temporaries holding the value of $\phi^S$, $\phi^L$, $\phi^V$, $\phi^C$, and $\phi$ instructions with SSA variables.

1: proc CommitVarRecur($DI : \mathcal{INST}$) : bool begin
2: Changed := true
3: if $DI = \phi^S pv, var, val, curr$ then
4: Changed := CommitVar($DI, curr, var, UsefullRefs$)
5: If not present, insert the instruction: store $pv, val$
6: else if $DI = \phi^L pv, \langle var_1, val_1 \rangle, \ldots, \langle var_n, val_n \rangle$ then
7: for $i := 1$ to $n$ do
8: if CommitVar($DI, val_i, var_i, UsefullRefs$) then
9: Changed := true
10: else if $DI = \phi^V \langle cs_1, val_1 \rangle, \ldots, \langle cs_n, val_n \rangle$ ∨
11: $DI = \phi^C \langle P_1, val_1 \rangle, \ldots, \langle P_n, val_n \rangle$ then
12: var := PhiVar[InstToTemp($DI$)]
13: for $i := 1$ to $n$ do
14: if CommitVar(getPred($DI, i), val_i, var_i, UsefullRefs$) then
15: Changed := true
16: return Changed
17: end
18: proc getPred($I : \mathcal{INST}, i : Z$) : $\mathcal{INST}$ begin
19: if $I = \phi \langle BB_1, val_1 \rangle, \ldots, \langle BB_n, val_n \rangle$ then
20: return getTerminator($BB_i$)
21: else if $I = \phi^V \langle cs_1, val_1 \rangle, \ldots, \langle cs_n, val_n \rangle$ then
22: return getPrevInst(ProgPointToInst($ci_i$))
23: else if $I = \phi^C \langle P_1, val_1 \rangle, \ldots, \langle P_n, val_n \rangle$ then
24: return getEndInst($P_i$)
25: else
26: return InvalidInst
27: end

Procedure CommitVarRecur, presented in Algorithm 5.7 makes certain that temporaries holding the value of $\phi$, $\phi^S$, $\phi^L$, $\phi^V$, and $\phi^C$ instructions can be replaced. Let us assume that the temporary $%DI0$ holds the value of $DI$. On line 3, we ensure that if $%DI0$ holds the value of $\phi^S pv, var, val, curr$ then $%DI0$ can be replaced with a load of $var$. To enable this, we first have to make certain that $var$ is equal to $curr$ prior to $DI$ through a call to procedure CommitVar on line 4. Moreover, we must account for the impact of aliasing (i.e. $pv$ is equal to $var$) by introducing a store instruction assigning $val$ to $var$ after $DI$. Next, let us assume that $DI = \phi^L pv, \langle var_1, val_1 \rangle, \ldots, \langle var_n, val_n \rangle$. In
order to replace %DI0 with a load of pv, we must ensure that each variable var<sub>i</sub>, where 1 ≤ i ≤ n, is equal to val<sub>i</sub> at the program point of DI (InstToProgPoint(DI)). This is done by committing val<sub>i</sub> to var<sub>i</sub> at InstToProgPoint(DI) through a call to procedure CommitVar on line 8. Finally, let us assume that DI is a φ, φ<sup>V</sup>, or φ<sup>C</sup> instruction. In order to replace %DI0 with a load of its corresponding propagation variable var, we must be certain that var is equal to %DI0 at InstToProgPoint(DI). As such, on lines 10–14, we commit each of the incoming values merged, to var, at the appropriate predecessor.

Example 5.6 Committing SSA variables in Figure 5.2(c)  
In Algorithm 5.2, after \( \mathcal{VS}[\text{TempToInst}(\%v7),\%v1] \) is mapped to @A.num, we call procedure CommitVar on line 24 and pass it TempToInst(\%v7), \%v1, @A.num, and UsefulRefs. Since pvar in procedure CommitVar is equal to an SSA variable (@A.num), we proceed to call procedure CommitVarRecur. However, TempToInst(Op) is a call instruction and as such, no additional store instructions are introduced by calling CommitVarRecur. Then, we proceed to execute lines 11–14 in Algorithm 5.5. Since the only SSA assignment that assigns \%v1 to @A.num (and reaches I) is ProgPointToInst(S1) on line 4 in Figure 5.2(c), we convert this SSA assignment to the store instruction on line 6 in Figure 5.2(d).

5.4.7 Converting ISSA Form Back to SSA Form  
Converting the IR to SSA form is done by visiting each entry in the \( \mathcal{VS} \) mapping. Let us assume that \( \mathcal{VS}[(I, Op)] = var \) and that the parent procedure of I is P. If var is a parameter or a return value, then we replace Op with var. Otherwise, we replace Op with %LI, a temporary holding the value of the load instruction LI, whose pointer value is var. When Op is not defined in procedure P (i.e. interprocedural reference) then LI is placed after the propagation point of Op to P. If I is a φ<sup>S</sup> instruction, then LI is placed after the store instruction inserted when committing the corresponding propagation variable.
of InstToTemp($I$) in Algorithm 5.7, line 5. Otherwise, $LI$ is inserted after the defining instruction of $Op$ (i.e. TempToInst($Op$)).

Note that when all entries in $\mathcal{VS}$ are replaced, redundant interprocedural references from an instruction $I$ to a temporary $Op$ may still exist. When this occurs, we replace the reference to $Op$ at $I$ with the \textit{undefined constant}. We also remove $\phi^S$, $\phi^L$, $\phi^V$, and $\phi^C$ instructions as well as reference to the temporaries holding their value. Finally, we remove all SSA assignments.

Beyond this, we use the compiler infrastructure to further clean up the IR. The passes applied after the out-of-ISSA translation, which were described in Section 5.3.2 will remove unused instructions, parameters, and program variables.

5.4.8 Impact of Transformations and Out-of-ISSA Translation

When applying the transformations in Section 5.3 and the out-of-ISSA translation presented in this section, the resulting SSA form is shown in Figure 5.2(d). The first transformation applied converts the stack allocated structures whose addresses are assigned to the temporaries $%vA$ and $%vA1$ to global variables. As described in Section 5.3.2, we apply a number of passes on the ISSA form. One benefit is that the parameter $l$ in procedure \textit{getCoefs} is folded to the constant 0. At the end, we clean up the IR with the passes applied after the out-of-ISSA translation.

There are a number of improvements from the various passes in the resulting SSA form (shown in Figure 5.2(d)). First, note that the store and load instructions used to pass structures across procedures have been removed. This occurs because we are propagating the temporaries $%v0$, $%v1$, and $%v2$ that are defined in procedure \textit{init} using the global variable $A$. This benefit was obtained because ISSA form has the notion of an interprocedural value. Using this notion, we folded the $\phi^V$ instructions that corresponded to the parameters $%a10$, $%a11$, and $%a12$ that passed the elements of structure $A$ into procedure \textit{getCoefs}. Then, we were able to determine that these values can be
propagated using the global structure $A$. Due to this transformation, we were also able to eliminate structure parameters from the procedure \textit{getCoefs}. Lastly, since structure $A$ in Figure 5.2(a) was converted to a global variable, the address of $A$ did not have to be passed to procedure \textit{init}; eliminating another parameter.

5.5 Experimental Evaluation

Our implementation was done in the LLVM infrastructure [30]. Prior to generating ISSA form, standard LLVM passes that performed constant propagation, dead code removal, and SSA form construction were applied. Afterwards, as discussed in Section 5.3, we apply various passes that leverage ISSA form, followed by the out-of-ISSA translation, and then additional passes that clean up the code. The \textit{baseline passes} consist of all passes that are applied prior to constructing ISSA form and after translating out of ISSA. Then the IR is linked and input to the \textit{llc} backend which in part utilizes GCC ($-O3$) to generate machine code. The experiments were performed on an Intel CORE 2 Duo 1.66 GHz processor, with 4 GB memory and running 64-bit ubuntu.

Our evaluation was on a set of MediaBench [31] and SPECINT2000 [1] benchmarks. We chose to study MediaBench benchmarks due to the absence of recursive data structures (recursive data structure fields are not SSA variables) and because we believe ISSA would be highly useful for media applications. To obtain longer runtimes, the reference input was changed: for \textit{G721} and \textit{GSM}, we repeated the original track (clinton.pcm) multiple times; for \textit{MPEG}, a 10 MB movie file was used; and lastly, \textit{JPEG} encoded a 1 MB picture six times. In addition, we evaluate our work on a subset of the SPECINT2000 benchmarks, with the exception of the benchmarks \textit{176.gcc}, \textit{252.eon}, \textit{253.perlbmk}, and \textit{255.vortex}. As described in Section 4.8.1, the benchmarks \textit{255.vortex} and \textit{176.gcc} were not included because the ISSA form construction algorithm does not scale to them. We did not include the benchmarks \textit{252.eon} and \textit{253.perlbmk} due to various issues outlined in
Section 4.8.2. When studying SPECINT2000 benchmarks, we used the reference input.

In the LLVM infrastructure, program runtime is evaluated by running the benchmark three times and taking the minimum user+system time of these executions. In this approach, each benchmark has three “opportunities” to attain its optimal execution time. The program runtimes reported in this thesis were derived using this approach.

The compiler optimizations applied on ISSA IR, namely constant propagation, dead code removal, and common subexpression elimination are commonly used by compilers. Typically, compilers apply these optimizations multiple times, to clean up the IR and simplify the implementation of other optimizations. In our implementation, we rely on the constant propagation pass to fold instructions so that each SSA variable is only accessed using a unique (constant) object. We applied these optimizations when evaluating both the proposed out-of-ISSA translation and the naive extensions. Due to the higher number of instructions in ISSA form, the runtime of these passes was slightly longer.

In Table 5.1, we present the time it takes to generate ISSA form and then convert back using the out-of-ISSA translation pass, in the second and third columns, respectively. As indicated, the out-of-ISSA translation pass is much faster than the ISSA pass, scaling to all analyzed benchmarks. This demonstrates that transforming the IR back to SSA form is always feasible using our approach.

Table 5.1 also contains the program runtime when comparing the baseline passes in LLVM [30] with our system. A performance improvement can be observed in most benchmarks, which demonstrates that performance can be maintained or even improved while switching between IR forms. The performance improvement can be attributed to constant folding, common subexpression elimination, removal of load and store instructions, and more efficient value propagation.

Moreover, in the column labeled “Adapt” in Table 5.1, we present the code performance of our naive out-of-ISSA translation, which is based on the out-of-SSA translation by Briggs [11]. In this translation, scalar global variables were used to propagate values
### Table 5.1: Compilation and program runtime in seconds.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Passes</th>
<th>Program</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ISSA</td>
<td>Out-Of-</td>
<td>LLVM</td>
<td>LLVM+</td>
<td>Speedup</td>
<td>Adapt</td>
<td>Speedup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ISSA</td>
<td>LLVM</td>
<td>LLVM+</td>
<td>Factor</td>
<td>Factor</td>
<td>Factor</td>
</tr>
<tr>
<td>G721</td>
<td>0.3</td>
<td>0.1</td>
<td>19.85</td>
<td>19.86</td>
<td>1.00</td>
<td>22.47</td>
<td>0.88</td>
</tr>
<tr>
<td>GSM</td>
<td>1.3</td>
<td>0.2</td>
<td>6.62</td>
<td>6.51</td>
<td>1.02</td>
<td>7.49</td>
<td>0.88</td>
</tr>
<tr>
<td>JPEG</td>
<td>10.8</td>
<td>1.5</td>
<td>1.47</td>
<td>1.40</td>
<td>1.05</td>
<td>3.94</td>
<td>0.37</td>
</tr>
<tr>
<td>MPEG2</td>
<td>2.9</td>
<td>0.9</td>
<td>21.87</td>
<td>21.00</td>
<td>1.04</td>
<td>36.41</td>
<td>0.60</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0.8</td>
<td>0.3</td>
<td>171.53</td>
<td>169.37</td>
<td>1.01</td>
<td>240.46</td>
<td>0.71</td>
</tr>
<tr>
<td>175.vpr</td>
<td>6.0</td>
<td>1.3</td>
<td>67.99</td>
<td>62.24</td>
<td>1.09</td>
<td>70.82</td>
<td>0.96</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1.3</td>
<td>0.1</td>
<td>226.10</td>
<td>221.29</td>
<td>1.02</td>
<td>271.90</td>
<td>0.83</td>
</tr>
<tr>
<td>186.crafty</td>
<td>8.3</td>
<td>2.5</td>
<td>64.32</td>
<td>63.58</td>
<td>1.01</td>
<td>210.83</td>
<td>0.31</td>
</tr>
<tr>
<td>197.parser</td>
<td>21.5</td>
<td>3.1</td>
<td>232.41</td>
<td>231.54</td>
<td>1.00</td>
<td>508.52</td>
<td>0.46</td>
</tr>
<tr>
<td>254.gap</td>
<td>91.4</td>
<td>5.2</td>
<td>59.88</td>
<td>59.85</td>
<td>1.00</td>
<td>92.16</td>
<td>0.65</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0.8</td>
<td>0.2</td>
<td>138.98</td>
<td>133.79</td>
<td>1.04</td>
<td>157.55</td>
<td>0.88</td>
</tr>
<tr>
<td>300.twolf</td>
<td>39.7</td>
<td>3.0</td>
<td>198.38</td>
<td>196.20</td>
<td>1.01</td>
<td>463.67</td>
<td>0.43</td>
</tr>
</tbody>
</table>

The program runtime is provided for the LLVM baseline (LLVM column), the LLVM baseline with the passes described in this report (LLVM+ column), and the LLVM baseline with the out-of-SSA algorithm adapted for ISSA IR where globals are used instead of locals (Adapt column). In the Speedup Factor columns, we provide the program performance improvement for the IR generated using the two out-of-ISSA translation algorithms against the LLVM baseline (i.e. divide program runtime numbers).

Between procedures and replace $\phi^V$ and $\phi^C$ instructions. Note the runtime comparison with the LLVM baseline in the adjacent column. As indicated, a slowdown was observed, primarily due to an increase in the number of copy instructions that were inserted to replace $\phi$, $\phi^V$, and $\phi^C$ instructions.

In order to provide more insight into the performance improvement, we illustrate the impact of our passes on the IR when compared to the LLVM baseline. In Figure 5.4, we illustrate the percentage reduction in the number of arguments, store instructions, and load instructions as well as the kind of SSA variables handled. As indicated in Figure 5.4(a), a large number of SSA variables are non-globals. This accentuates the impact of the storage-remap transformation. In the benchmark 175.vpr in SPECINT2000, we observed the highest speedup. In this benchmark, a number of structures allocated on the stack are passed across call sites as parameters. During the storage-remap pass these
Figure 5.4: In Figure 5.4(a) we illustrate the distribution of SSA variables into scalar globals, non-scalar globals, and stack and heap allocated variables. In Figure 5.4(b), we illustrate the percentage decrease in the number of arguments that occurs when we use our passes (“LLVM+”) in addition to the LLVM baseline (“LLVM”).
Figure 5.4: In the subfigures above, we illustrate the percentage decrease in the number of store instructions and load instructions that occurs when we use our passes (“LLVM+”) in addition to the LLVM baseline (“LLVM”). (Continued)
structures are converted into global variables. This enables subsequent LLVM passes to remove 31% more arguments, 16% more store instructions, and 18% more load instructions. The second largest speedup occurred for JPEG, where the storage-remap transformation allowed us to eliminate parameters and as indicated in Table 5.2, fold a very large number of pointer arithmetic instructions. The large increase in the number of pointer arithmetic instructions folded allowed us to reduce the number of propagated pointer values, thus contributing to performance improvement. We suspect that a side benefit would be a reduction in register pressure and spilling.

5.5.1 Constant Propagation

We implemented a pass that performs constant propagation and dead code removal using ISSA, based on the Wegman and Zadeck algorithm [51]. The constant propagation pass was further improved by leveraging the $\phi^V$ and $\phi^C$ instructions to evaluate a procedure’s instructions under the call sites that invoke it. Moreover, we examined the pointer value at indirect call sites in order to infer values of temporaries based on the target procedure called.

In Table 5.2, we show the effectiveness of the ISSA-based constant propagation in comparison to the LLVM [30] constant propagation (-instcombine, -ipconstprop). When summarizing the constant folded instructions on all benchmarks, excluding all instructions folded during dereference conversion and copy propagation, we noted a 10.8% improvement on top of the LLVM passes.

5.5.2 Dead Code Removal

In Table 5.3, we present the number of basic blocks left after applying the LLVM baseline passes and after the LLVM baseline passes along with our proposed passes are applied (on ISSA form). As indicated, 1.8% more basic blocks are removed using the proposed passes, because we folded additional branches and removed unreachable code. Moreover,
we identified procedures that will exit the program when invoked and eliminated code that follows call sites which target these procedures.

### 5.5.3 Common Subexpression Elimination

In Table 5.4, we provide the number of instructions removed when applying the common subexpression elimination pass in the LLVM infrastructure over SSA and ISSA form IR. Note that when run on ISSA form IR, the common subexpression elimination pass removes 42.9% more instructions than for SSA. We examined the results for a number of benchmarks and noted that the improvements were primarily due to resolving load instructions of SSA variables to their definitions.
Table 5.3: Number of basic blocks left after baseline passes are applied (column labelled LLVM) and after LLVM passes along with our proposed passes are applied on ISSA form (column labelled LLVM+)

### 5.6 Summary

The out-of-ISSA translation poses a number of challenges and optimization opportunities. In this chapter, we showed that a naive extension of out-of-SSA translation algorithms, which does not address these challenges, outputs code whose runtime was 1.5 times slower than the LLVM baseline. To address this problem, we propose and validate an out-of-ISSA translation. We demonstrate that converting the IR to ISSA form and back using our proposed algorithm reduces the number of procedure parameters, load instructions, and stores instructions due to more efficient value propagation across procedures. Along with a set of standard optimizations, this results in program performance improvement over the LLVM infrastructure. Based on our study, we believe that our key strategy, using ISSA form on certain client applications and translating out of ISSA form to accommodate unsupported compiler passes, paves a path towards integrating ISSA form into compilers.
Chapter 5. Out-of-ISSA Translation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Instructions Removed</th>
<th>△</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SSA</td>
<td>ISSA</td>
<td></td>
</tr>
<tr>
<td>G721</td>
<td>64</td>
<td>69</td>
<td>5</td>
</tr>
<tr>
<td>GSM</td>
<td>331</td>
<td>450</td>
<td>119</td>
</tr>
<tr>
<td>JPEG</td>
<td>1284</td>
<td>1407</td>
<td>123</td>
</tr>
<tr>
<td>MPEG2</td>
<td>250</td>
<td>532</td>
<td>282</td>
</tr>
<tr>
<td>164.gzip</td>
<td>190</td>
<td>258</td>
<td>68</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1737</td>
<td>2192</td>
<td>455</td>
</tr>
<tr>
<td>181.mcf</td>
<td>110</td>
<td>140</td>
<td>30</td>
</tr>
<tr>
<td>186.crafty</td>
<td>1817</td>
<td>2024</td>
<td>207</td>
</tr>
<tr>
<td>197.parser</td>
<td>1084</td>
<td>2631</td>
<td>1547</td>
</tr>
<tr>
<td>254.gap</td>
<td>6455</td>
<td>8013</td>
<td>1558</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>287</td>
<td>351</td>
<td>64</td>
</tr>
<tr>
<td>300.twolf</td>
<td>3569</td>
<td>5669</td>
<td>2100</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: The number of instructions removed when running the common subexpression elimination pass on SSA (column labelled *SSA*) and ISSA (column labelled *ISSA*) form. When run on ISSA form, the common subexpression elimination pass removes 42.9% more instructions.
Chapter 6

ISSA-Based Interprocedural Induction Variable Analysis

6.1 Introduction

Induction variable analysis computes the evolution of variables inside a loop and represents it using a mathematical expression. Because computing the evolution is crucial for a vast number of analyses and optimizations, the induction variable analysis is a critical component in modern compilers. Loop transformation and parallelization algorithms depend on the induction variable analysis to compute the trip counts and loop carried dependencies. Induction variable analysis is also used for strength reduction, constant folding, and determining the bit-width of expressions.

Previous induction variable analysis algorithms were largely confined within the scope of procedures [25, 48, 52]. To our knowledge, the only exception is the interprocedural induction variable analysis proposed by Tang and Yew [47], which computes the evolution of the parameters in recursive procedures. While a pioneering effort in which the first interprocedural induction variable analysis algorithm was described, neither the complexity nor the benefits were quantified with benchmarks.
Given modern intraprocedural induction variable analysis algorithms leverage $\phi$ instructions in SSA form, naturally, the following questions can be asked:

- Can the interprocedural induction variable analysis benefit from ISSA?
- How many more induction variables can we identify?
- Can the interprocedural induction variable analysis enable new applications?

In this chapter, we attempt to answer these questions by using ISSA form to extend the induction variable analysis interprocedurally and make the following contributions:

- We show that the intraprocedural algorithm can be trivially extended into its interprocedural counterpart by using ISSA form. Our implementation adds only 370 lines of C++ code to the scalar evolution pass in the LLVM infrastructure.

- For the first time, we evaluate the benefit of an interprocedural induction variable analysis on large-scale benchmarks, including SPECINT2000 [1] and MediaBench [31]. Compared to the LLVM [30] infrastructure, in a consistent manner (for almost every benchmark), we observed an increase of 14.4% and 49.1% in the number of linear and monotonic induction variables, respectively. Moreover, we identified 1.1 and 2.6 times more constant and loop invariant trip counts, respectively.

- We demonstrate a new application of the induction variable analysis, in which we leverage ISSA to compute the file-position evolution. This can be used to compute file-access dependencies, parallelize loops with file accesses, and derive loop invariant trip counts.

The rest of this chapter is organized as follows. Section 6.2 reviews prior work on induction variable analysis. In Section 6.3, we motivate the interprocedural induction variable analysis. In Section 6.4, we describe the LLVM [30] induction variable analysis, which we extend to ISSA form. Section 6.5 presents our ISSA-based interprocedural
6.2 Background and Related Work

The value of certain program variables at a given loop iteration can be represented using a recurrence relation, in which each term in a sequence is defined as a function of preceding terms. We refer to such program variables as induction variables. For instance, the variable \( i \) in Figure 6.1(a) is an induction variable because it is incremented by 1 on every loop iteration. Note that induction variables have a predictable evolution, since we can determine and represent the change in their value. An induction variable analysis algorithm. Section 6.6 contains case studies and in Section 6.7, we provide an experimental evaluation. A summary is provided in Section 6.8.
will identify induction variables and compute a recurrence relation for them. The SSA form for the C source code in Figure 6.1(a) is shown in Figure 6.1(b). Since the variable \( i \) is assigned within the loop, a \( \phi \) instruction defining \( %i0 \) is inserted in the loop header.

To determine whether \( %i0 \) is an induction variable, we derive its recurrence relation. In our case, this is done by expressing \( %i1 \), which is the value of \( %i0 \) in the next iteration as a function of: loop invariants, other induction variables, and the value of \( %i0 \) in the current iteration. When doing this, we note that \( %i0_{\text{next iter}} = %i0_{\text{curr iter}} + 1 \), where \( %i0_{\text{next iter}} \) is the value of \( %i0 \) in the next loop iteration and \( %i0_{\text{curr iter}} \) is the value of \( %i0 \) in this iteration. This indicates that \( %i0 \) is incremented by 1 on every iteration of the loop and as such, we conclude that \( %i0 \) is an induction variable.

Initially, induction variable analysis algorithms focused on identifying linear induction variables, which are incremented by a loop invariant value on every loop iteration. As such, the value of linear induction variables at each loop iteration can be derived using a linear function. Wolfe [52] proposed the use of SSA to identify general induction variables. This includes polynomial induction variables, whose value is a polynomial function of the loop iteration as well as monotonic induction variables, whose value is either increasing, decreasing, non-increasing, or non-decreasing after each loop iteration. In the example in Figure 6.1(a), the variable \( j \) is a monotonically non-decreasing induction variable, since on every iteration it is either incremented by 2 or does not change. Gerlek et al. [25] implemented an induction variable analysis based on Wolfe’s algorithm and showed that on a wide set of Fortran benchmarks, less than 2% of subscript expressions are non-linear induction variables. Haghighat and Polychronopoulos [27] review work on general induction variables and expressions using a symbolic analysis.

In SSA form, the uses of scalar stack variables whose address is not taken are replaced with temporaries that are assigned just once. These temporaries can be thought of as scalars that hold the value computed by their defining instruction. Given this observation, the induction variable analysis is referred to as the scalar evolution analysis in
various compilers [26, 30]. The scalar evolution analysis accepts a temporary as input and computes its recurrence relation.

We can represent the value of an induction variable at each loop iteration using a polynomial function. However, deriving the coefficients of a polynomial function can be relatively difficult when the recurrence relation contains symbols, such as loop invariant temporaries. To overcome this difficulty, in both GCC [26, 38] and LLVM [30], the recurrence relation is represented by using chains of recurrences, which is a representation of closed-form functions proposed by Bachmann, Wang, and Zima [6].

A detailed introduction to using chains of recurrences for the induction variable analysis is provided by Pop et al. [37] and below we cover the basic concept by assuming the depth of the loop $L$ is 1. A chain of recurrence function $\text{CR}(L, v_1, v_2)$ has an initial value $v_1$ and an increment of $v_2$. For instance, the value of $\text{CR}(L, 8, 5)$ in the first three iterations of $L$ is 8, $8 + 5 = 13$, and $8 + (5 + 5) = 18$. Given this, the value of $\text{CR}(L, 3, \text{CR}(L, 8, 5))$ in the first four iterations of $L$ is 3, $3 + 8 = 11$, $3 + (8 + 13) = 24$, and $3 + (8 + 13 + 18) = 42$. Using the Newton interpolation series, we can efficiently compute the value of a chains of recurrence function at a given loop iteration. Leveraging chains of recurrences for the induction variable analysis and dependency testing was first proposed by van Engelen et al. [8, 42, 48].

Tang and Yew [47] described an interprocedural induction variable analysis. Rather than deriving the evolution of program variables between loop iterations, it computes the evolution between invocations of recursive procedures. In their algorithm, the values passed through certain parameters in recursive procedures were analyzed to identify a recurrence relation; however, neither the benefit nor the complexity were quantified with benchmarks.

Unlike our work, all previous approaches did not identify induction variables in globals, singleton heap locations, or structure fields. Furthermore, out of all these previous techniques, only Tang and Yew traced values interprocedurally, but only through param-
Chapter 6. ISSA-Based Interprocedural Induction Variable Analysis

parameters in recursive procedures.

6.3 Motivation

To illustrate the interprocedural induction variable analysis, we use the C source code presented in Figure 6.2(a), whose ISSA form is shown in Figure 6.2(b). Note that procedure Next is called twice, at call sites CI1 and CI2. Moreover, the global variable index is used within procedure Next. Therefore, we insert a $\phi$ instruction, whose result is held in the temporary $%v0$, on line 2 in Figure 6.2(b) to merge the two value of index at CI1 and CI2. In procedure Next, a call is made to procedure getc to read the next byte in the file and afterwards, this byte is written to element index in the array $b0$. Then, we increment $%v0$ (i.e. value of index) by 1 in the addition instruction whose result is held in $%v3$ and after this, we return from procedure Next. Because the global variable index is included in the set of SSA variables, we insert the $\phi$ instruction whose result is held in the temporary $%v6$ in Figure 6.2(b), line 19. Note that this $\phi$ instruction is equal to 1 when entering BB2 from BB0. We were able to determine that index is equal to 1 when entering BB2 from BB0 using the context-sensitive constant propagation.

In the example shown in Figure 6.2(a), there are two predictable evolutions in the loop shown in procedure main that traditional induction variable analysis algorithms do not identify. First, index is a linear induction variable with stride 1. Second, the file position of $pn$ is moved to the next byte on every iteration of the loop. Identifying these two induction variables can be leveraged to parallelize this loop (in main).

In many benchmarks, data from files are processed in chunks and the result is outputted to other files or devices, sequentially. Using the file-position evolution we can determine the sections of code affected by regions (or bytes) of the file. This information can be used to create multiple versions of a program section, that are optimized for specific values that a region of the file has. Moreover, we can identify and compute file-access
dependencies. This can be used to parallelize loops that contain I/O operations, because we can compute the dependencies and the sections (index) of the file that are accessed at a given iteration. Beyond these benefits, as shown in Section 6.6.4, evaluating the file position evolution can help us identify and compute additional loop invariant trip counts.

An ISSA-based interprocedural induction variable analysis has a number of benefits. First, modern compilers [26, 30] have an SSA-based induction variable analysis implementation, which can easily be adapted to ISSA form (370 lines of C++ code on top of the LLVM implementation). Second, we can make the analysis context-sensitive in an efficient manner. In ISSA form, we handle more SSA variables and can derive the call
graph path between definitions and uses. Using the call graph path, we can evaluate $\phi^V$ instructions for the single relevant context needed, rather than explore redundant paths. Context sensitivity is useful when we trace the recurrence relation across procedures. For instance, in Figure 6.2(b), since $%v0$ could be equal to 0 or $%v6$ then $%v3 = %v0 + 1$ could be equal to 1 or $%v6 + 1$.

With a context-sensitive induction variable analysis we can determine that $%v6$ is incremented by 1 within the loop on lines 14–21 since $%v0 = %v6$ and $%v3 = %v6 + 1$ under the context $CI2$ and as such, we can derive the recurrence relation $%v6_{nextiter} = %v6_{curriter} + 1$.

### 6.4 Baseline Induction Variable Analysis

Our induction variable analysis is based on the LLVM implementation [30], which is located in the scalar evolution pass and reviewed in Section 6.4.1. In LLVM, the induction variable analysis accepts a value as an input and computes a Scalar Evolution Expression (SCEV) for it. Currently, the LLVM implementation identifies only linear and polynomial induction variables. We extended it to identify monotonic induction variables, as described in Section 6.4.2.

#### 6.4.1 LLVM Scalar Evolution Pass

In order to describe the LLVM scalar evolution analysis, we leverage the framework presented in Section 2.3 and add to it the following:

- $C\langle cnst \rangle$: SCEV representing a constant $cnst$.
- $I\langle %I0 \rangle$: SCEV representing the temporary $%I0$.
- $+\langle S_1, S_2 \rangle$: SCEV representing the addition of the SCEVs $S_1$ and $S_2$.
- $CR\langle L, S_1, S_2 \rangle$: SCEV representing a polynomial induction variable for loop $L$ in chains of recurrences form. Its initial value is the SCEV $S_1$ and its increment is the SCEV
If $S_2$ is loop invariant, then the SCEV represents a linear induction variable.

The procedure $getSCEV$, whose pseudocode is given in Algorithm 6.1, computes the SCEV for $TheVal$, which is either a temporary or a constant. In the base case, $TheVal$ is a constant and we simply return $C\langle TheVal \rangle$ on line 4. Otherwise, $TheVal$ must be a temporary $%I0$, which holds the value of instruction $I$. In procedure $getSCEV$, we first query the hashtable ($MP$) and return the corresponding SCEV for $%I0$ if it is found. Next, we select the routine that computes the SCEV by matching $I$ to various instructions. In Algorithm 6.1, we focus on $\phi$ instructions located in the loop header. The loop in which an instruction is located can be derived using the function $getParentLoop$. A $\phi$ instruction in the loop header merges the value of a given variable when entering the loop (from the entry basic block) and after each iteration is completed (from the latch basic block).

When $I$ is a $\phi$ instruction located in the loop header of loop $L$, we proceed by entering the SCEV $I\langle %I0 \rangle$ into $MP[ %I0 ]$. As stated in Section 6.2, the value of $%I0$ in the next iteration of loop $L$ is equal to $V_{inc}$, the value coming into the $\phi$ instruction from the latch basic block. Hence, we derive the recurrence relation for $%I0$ by computing the SCEV for $V_{inc}$ using the procedure $getSCEV$. Note that calls to $getSCEV( %I0 )$ executed while computing the SCEV for $V_{inc}$ return $I\langle %I0 \rangle$, which denotes the value of $%I0$ in the current iteration. Once the recurrence relation is computed in the SCEV $S_{inc}$, we analyze $S_{inc}$ in order to determine whether $%I0$ is an induction variable. The SCEV for the temporary $%I0$ is a polynomial induction variable under two conditions:

1. If $S_{inc}$ indicates that on each iteration of loop $L$, we add to $I\langle %I0 \rangle$ an SCEV $S_{exp}$ that is either loop invariant in relation to $L$ or a polynomial induction variable whose loop is $L$ (i.e. the added SCEV is $CR(L, \ldots)$).

When this condition is satisfied, $Result$ is assigned a polynomial induction variable whose increment is $S_{exp}$. In the pseudocode shown in Algorithm 6.1, we test for
this condition and assign the SCEV to Result on lines 13–15

2. If \( S_{inc} \) is equal to a linear induction variable \( CR_{L,C\{cbase\},C\{cinc\}} \), the SCEV for \( V_{start} \) is \( C\{cinit\} \), and the difference between \( cbase \) and \( cinc \) equals \( cinit \).

In this scenario, on each iteration \( i > 1 \), \( \%I_0 \) is equal to the value of \( CR_{L,C\{cbase\},C\{cinc\}} \) at iteration \( i - 1 \). Since \( cinit \) is equal to \( \%I_0 \) in the first iteration and \( cbase - cinc = cinit \), therefore the value of \( \%I_0 \) at each iteration is equal to \( CR_{L,C\{cinit\},C\{cinc\}} \).

**Example 6.1 Computing the induction variable for \( \%i_0 \) in Figure 6.1(b)**

First, note \( \%i_0 \) holds the value computed by a \( \phi \) instruction located in the loop header. When calling procedure \( getSCEV \), we enter \( I\{\%i_0\} \) into \( MP[\%i_0] \) on line 12. Then, we call procedure \( getSCEV \) and pass it \( \%i_1 \), which is a temporary holding the value of an addition instruction whose operands are \( \%i_0 \) and 1. For addition instructions, we call \( getSCEV \) on each of their operands. When calling \( getSCEV(\%i_0) \) we note that \( MP[\%i_0] = I\{\%i_0\} \) on line 7 and return \( I\{\%i_0\} \). Afterwards, we call \( getSCEV(1) \) that returns the SCEV \( C\{1\} \) on line 4. Then we add these two SCEVs together yielding \( +\{I\{\%i_0\},C\{1\}\} \). Since this pattern matches the one on line 13 in Algorithm 6.1 and \( C\{1\} \) is loop invariant, then we compute the SCEV for the value of \( \%i_0 \) when entering the loop, which is \( C\{0\} \). This leads us to execute line 15 in the pseudocode that assigns Result the SCEV \( CR\{L,C\{0\},C\{1\}\} \), which is a linear induction variable in loop \( L \) with an initial value of 0 and an increment of 1.

Other instructions are also processed by invoking \( getSCEV \) recursively. However, cycles can only occur when processing \( \phi \) instructions and as such, there is no need to insert \( I \) into the map as was done on line 12. During the derivation of SCEVs, folding and simplification are applied.

**6.4.2 Monotonic Induction Variables**
Algorithm 6.1  Procedure getSCEV. It accepts as input the value \( V \) and returns its SCEV. Recall that procedure TempToInst is described in Section 2.3. It accepts a temporary as input and returns the instruction whose value it holds.

1: \( \mathcal{MP} : \mathcal{T} \mathcal{MP} \mapsto \text{SCEV} := \emptyset \)
2: proc getSCEV(TheVal : VAL) : SCEV begin

Require:  TheVal is either a constant or a temporary
3: if TheVal is a constant then
4: return \( C\langle \text{TheVal} \rangle \)
5: make certain TheVal is a temporary \( %I0 \)
6: if \( \mathcal{MP}[%I0] \neq \emptyset \) then
7: return \( \mathcal{MP}[%I0] \)
8: Result := \( I\langle %I0 \rangle \)
9: \( I := \text{TempToInst}(%I0) \)
10: \( L := \text{getParentLoop}(I) \)
11: if \( I = \phi \langle \text{latch}, V_{\text{inc}}, \langle \text{entry}, V_{\text{start}} \rangle \rangle \) and \( \text{getParent}(I) = \text{header}(L) \) then
12: \( \mathcal{MP}[%I0] := \text{Result} \)
13: if getSCEV(\( V_{\text{inc}} \)) = \( +\langle I\langle %I0 \rangle, S_{\text{exp}} \rangle \) then
14: if isLoopInvariant(\( S_{\text{exp}}, L \) \( ) \lor S_{\text{exp}} = \text{CR}(L, \ldots) \) then
15: Result := \( \text{CR}(L, \text{getSCEV}(V_{\text{start}}), S_{\text{exp}}) \)
16: else if getSCEV(\( V_{\text{inc}} \)) = \( \text{CR}(L, C\langle \text{cbase} \rangle, C\langle \text{cinc} \rangle) \) then
17: \( S_{\text{init}} := \text{getSCEV}(V_{\text{start}}) \)
18: if \( S_{\text{init}} = C\langle \text{cinit} \rangle \land \text{cbase} - \text{cinc} = \text{cinit} \) then
19: Result := \( \text{CR}(L, C\langle \text{cinit} \rangle, C\langle \text{cinc} \rangle) \)
20: else if \( I = \phi \langle BB_1, val_1 \rangle, \ldots, \langle BB_n, val_n \rangle \) then
21: Result := processPHI(%I0)
22: else
23: Result := getSCEVNonPHI(%I0)
24: \( \mathcal{MP}[%I0] := \text{Result} \)
25: return Result
26: end

In order to support monotonic induction variables, we made three changes to the LLVM infrastructure. First, we defined the \( \mathcal{NN} \) class to represent non-negative values of a given type. To simplify the presentation, in this chapter we assume that the \( \mathcal{NN} \) class also represents a SCEV. In our notation, \( \mathcal{NN}\langle \rangle \) denotes a \( \mathcal{NN} \) SCEV. Second, we added additional code that supports constant folding for \( \mathcal{NN} \) objects as well as code to support simplification of \( \mathcal{NN} \) objects in SCEV operations. Finally, we compute the SCEVs for \( \phi \) instructions that are not located in the loop header by calling procedure processPHI on line 21 in Algorithm 6.1. In Algorithm 6.2, we present procedure processPHI, which
Algorithm 6.2 Procedure processPHI. It computes the SCEV for a temporary defined by a \( \phi \) instruction.

1: proc processPHI(\%I0 : TMP) : SCEV begin

Require: \%I0 is defined by a \( \phi \) instruction: \( \phi(BB_1, val_1), \ldots, (BB_n, val_n) \)

2: \( \mathcal{M}[\%I0] := \mathcal{I}(\%I0) \)
3: Result := getSCEV(val_1)
4: for \( i := 2 \) to \( n \) do
5:  \( S := \) getSCEV(val_i)
6:  if Result \neq S then
7:   if isGEQToZero(S) \& isGEQToZero(Result) then
8:     Result := \( \mathcal{N}\mathcal{N}(\emptyset) \)
9:   else
10:      Result := collapseIntoMonAdd(Result)
11:      \( S := \) collapseIntoMonAdd(S)
12:      if Result \neq S then
13:        return \( \mathcal{I}(\%I0) \)
14:   return Result

end

proc isGEQToZero(S : SCEV) : bool begin
17: return S = \( \mathcal{N}\mathcal{N}(\emptyset) \) \& (S = C(const) \& const \geq 0)
end

proc collapseIntoMonAdd(S : SCEV) : SCEV begin
21: if S = \( \mathcal{I}(\%I0) \) then
22:   return +\( \langle S, \mathcal{N}\mathcal{N}(\emptyset) \rangle \)
23: else if S = +\( \langle \mathcal{I}(\%I0), T_1, \ldots, T_n \rangle \) \& isGEQToZero(T_i), 1 \leq i \leq n then
24:   return +\( \langle \mathcal{I}(\%I0), \mathcal{N}\mathcal{N}(\emptyset) \rangle \)
25: else
26: return S
end

accepts as input a temporary \%I0 that is defined by a \( \phi \) instruction I and returns its SCEV.

If all the incoming values of I are greater or equal to 0, then \( \mathcal{N}\mathcal{N}(\emptyset) \) is returned by Algorithm 6.2. When each incoming value is greater or equal to a SCEV S, then +\( \langle S, \mathcal{N}\mathcal{N}(\emptyset) \rangle \) is returned. Otherwise, \( \mathcal{I}(\%I0) \) is returned. As a result of this change, the operands of SCEVs can be \( \mathcal{N}\mathcal{N} \) objects. A monotonic induction variable will be a linear induction variable whose increment is \( \mathcal{N}\mathcal{N} \).

Example 6.2 Computing the induction variable for \%j0 in Figure 6.1(b)
First, note \( %j0 \) is assigned a \( \phi \) instruction in the loop header, hence we enter \( I(\%j0) \) into \( MP[\%j0] \) on line 12 in Algorithm 6.1 and we call procedure \( \text{getSCEV} \) and pass it \( %j2 \), which is defined by a \( \phi \) instruction. We proceed to evaluate the operands of \( %j2 \) by calling \( \text{processPHI} \). In procedure \( \text{processPHI} \), we assign \( \text{Result} \) the SCEV for \( %j1 \) on line 3 in Algorithm 6.2, which is \( +\langle I(\%j0), C(2) \rangle \). Then, we enter the loop on lines 4–13, where we assign the SCEV for \( %j0 \), which is \( I(\%j0) \), to \( S \) on line 5. Since \( \text{Result} \) is not equal to \( S \) and we cannot prove that either of them is greater or equal to 0, then we try to collapse both \( \text{Result} \) and \( S \) into a monotonic addition SCEV. Note that both of these SCEVs either add 2 or nothing to \( %j0 \). Hence, the SCEV \( +\langle I(\%j0), \mathcal{N}(\mathcal{N}) \rangle \) is returned by the calls to procedure \( \text{collapseIntoMonAdd} \), when \( \text{Result} \) and \( S \) are passed as arguments. Hence, \( +\langle I(\%j0), \mathcal{N}(\mathcal{N}) \rangle \) will be returned by the \( \text{processPHI} \) call and the \( \text{getSCEV} \) call that computes the SCEV of \( %j1 \). Since this pattern matches the one on line 13 in Algorithm 6.1 and \( \mathcal{N}(\mathcal{N}) \) objects are loop invariant, we derive the SCEV \( CR(L, C(0), \mathcal{N}(\mathcal{N})) \) for \( %j0 \).

### 6.5 ISSA-Based Interprocedural Induction Variable Analysis

In this section, we present the ISSA-based interprocedural induction variable analysis. In Section 6.5.1, we present the predicated induction variable. In order to compute the recurrence relation across procedures, we leverage ISSA form. We describe how ISSA IR extensions are supported in Section 6.5.2. We describe how we leverage \( \phi^V \) and \( \phi^C \) instructions to compute the context-sensitive recurrence relation in Section 6.5.3. The file-position induction variable analysis is presented in Section 6.5.4.
6.5.1 Predicated Induction Variables

The evolution of a variable in a loop $L$ sometimes depends on a loop invariant condition. We define a predicated $SCEV$, $\triangle_{\text{LHS,RHS}}\langle L, T_S, F_S \rangle$ that represents a predicated value: if $LHS$ equals $RHS$ then its value is $T_S$, otherwise its value is $F_S$. The predicated $SCEV$ is constructed when processing $\phi^S$, $\phi^L$, and selection instructions in procedure $\text{getSCEVNonPHI}$, which is called from $\text{getSCEV}$ on line 23 in Algorithm 6.1. In Algorithm 6.3, we present the sections of procedure $\text{getSCEVNonPHI}$ that handle $\phi^S$, $\phi^L$, and selection instructions. The procedure $\text{getSCEVNonPHI}$ accepts as input a temporary $\%I_0$ defined by instruction $I$ and returns its $SCEV$.

**Algorithm 6.3** Sections of procedure $\text{getSCEVNonPHI}$ that handle $\phi^S$, $\phi^L$, and selection instructions and can return predicated $SCEVs$. Recall that procedure $\text{TempToInst}$ accepts a temporary as input and returns the instruction whose value it holds.

```
1: proc getSCEVNonPHI($\%I_0 : \mathcal{TMP}$) : SCEV begin
2:     $I := \text{TempToInst}(\%I_0)$
3:     $L := \text{getParentLoop}(I)$
4:     if $I = \phi^S \ pval, \ var, \ val, \ curr$ then
5:         if $\text{isLoopInvariant}(pval, L)$ then
6:             return $\triangle_{\text{pval, var}}\langle L, \text{getSCEV}(val), \text{getSCEV}(curr) \rangle$
7:         else if $I = \phi^L \ pval, \ \langle var_1, val_1 \rangle, \ \langle var_2, val_2 \rangle$ then
8:             if $\text{isLoopInvariant}(pval, L)$ then
9:                 return $\triangle_{\text{pval, var_1}}\langle L, \text{getSCEV}(val_1), \text{getSCEV}(val_2) \rangle$
10:            else if $I = \text{select} \ \%v_0, \ val_1, \ val_2$ then
11:                $CmpI := \text{TempToInst}(\%v_0)$
12:                if $CmpI = \text{eq} \ v_1, \ v_2 \land \text{isLoopInvariant}(v_1, L) \land \text{isLoopInvariant}(v_2, L)$ then
13:                    return $\triangle_{\text{v_1, v_2}}\langle L, \text{getSCEV}(val_1), \text{getSCEV}(val_2) \rangle$
14:        ... 
15: end
```

If $\%I_0$ is defined by the instruction $\phi^S \ pval, var, val, curr$ (line 4) then $\%I_0$ will be equal to $val$ if $pval = var$ and to $curr$ otherwise. Hence, if $pval$ is loop invariant (loop $L$) then the $SCEV$ $\triangle_{\text{pval, var}}\langle L, \text{getSCEV}(val), \text{getSCEV}(curr) \rangle$ is returned. On line 7, we check whether $I$ is a $\phi^L$ instruction with two possible values and we return an $SCEV$ for $\%I_0$ if its pointer value is loop invariant. Finally, on line 10 we handle selection instructions that have an equality comparison operand $\%v_0$. If this is the case
and \(v_0\) is loop invariant then \(I_0\) can be represented using the predicated SCEV \(\triangle_{v_1,v_2}(L, \text{getSCEV}(val1), \text{getSCEV}(val2))\).

With this change, temporaries can be mapped to predicated SCEVs. Let us assume that a temporary \(I_0\) holds the value of a \(\phi\) instruction located in the loop header. If \(I_0\) is mapped to a predicated SCEV \(S = \triangle(L, CR(L, \ldots), I(I_0))\), then we say that \(I_0\) is a predicated induction variable.

### 6.5.2 Supporting ISSA Form

Our induction variable analysis was extended by adding cases to the pseudocode of procedure \(\text{getSCEV}\), shown in Algorithm 6.1. We describe the handling of ISSA form below:

**\(\phi^S\) and \(\phi^L\) instructions:** We first check whether a predicated SCEV can be assigned to them as outlined in Section 6.5.1, Algorithm 6.3. If not, then both \(\phi^S\) and \(\phi^L\) instructions are treated as merge points and are represented using \(I(I_0)\).

**\(\phi^V\) and \(\phi^C\) instructions:** By leveraging the \(\phi^V\) and \(\phi^C\) instructions, we evaluate SCEVs in a context-sensitive manner. Details regarding their support is provided in Section 6.5.3.

In summary, when a \(\phi^C\) instruction \(I\) whose value is held in the temporary \(I_0\) merges a single value \(V\), we map \(I_0\) to the SCEV for \(V\). Otherwise, the SCEV for \(I_0\) is \(I(I_0)\). Context sensitivity is achieved by restricting the possible values of \(\phi^V\) instructions using the call string.

The baseline scalar evolution pass assumes each operand of any instruction \(I\) is either a constant or a temporary that is defined within the same procedure as \(I\). Hence, many parts of the code were updated to account for operands defined in other procedures. For instance, the code used to determine whether a temporary is loop invariant was modified to handle temporaries defined in other procedures. When a temporary \(I_0\) is defined in
another procedure, we test whether its propagation point is loop invariant.

To perform these tests and update the call string we keep track of a number of things. First, when the initial call to $getSCEV$ is made, we identify the procedure in which the passed temporary is defined ($P$) and maintain a reference to it. This will be used to identify loop invariant values. Second, we keep track of the current loop $CurrL$, as we process instructions outside of $P$. When passing $getParentLoop$ an instruction whose parent procedure is $Q \neq P$, then $CurrL$ is returned. Finally, we maintain a reference to the instruction defining the temporary which is passed to the previous invocation of procedure $getSCEV$. It will be used to update the call string, as described in Section 6.5.3.

### 6.5.3 Context Sensitivity

The context-sensitive interprocedural induction variable analysis leverages ISSA form to determine the values of instructions under specific contexts. Context sensitivity is obtained by making only two significant changes to the baseline algorithm. First, we cache the values of temporaries under different contexts. Rather than mapping temporaries to their SCEV, we pair temporaries with their context and map each pair to the resulting SCEV. Second, we maintain $CS$ which is the call string that represents the call graph path to the instruction we are currently processing. Beyond being used to cache values (as the context), we use the call string to identify the context-specific value of $\phi^V$ instructions by matching their incoming points with the first entry in the current call string. More formally, the pseudocode for processing instruction $\%I0 := \phi^V \ldots, \langle cs, val \rangle, \ldots$ inside procedure $getSCEV$ (in Algorithm 6.1) where $|CS| > 0$ and $CS[|CS| - 1]$ equals $cs$ is:
pop(CS);
Result = getSCEV(val);
push(CS, cs);

With this change, we derive the context-specific values of instructions in the recurrence relation, thus transforming our induction variable analysis into a context-sensitive solution. The call string is extended when processing a $\phi^C$ instruction that merges a single value at call site $cs$. If $%I0 := \phi^C(\langle \ldots, val \rangle$ and $cs \notin CS$, then we proceed as follows:

push(CS, cs);
Result = getSCEV(val);
pop(CS);

If copy propagation is applied, then operands can be temporaries which are defined in other procedures, thus requiring us to update the call string accordingly. Let us assume that we currently process instruction $I$ in procedure $P$ and we want to compute the SCEV for one of its operands, a temporary $Op$ that is defined in procedure $Q \neq P$. In this scenario, we call procedure $updateCallString$, presented in Algorithm 6.4, which derives the call graph path $CS_{new}$ from $I$ to $Op$. If $Op$ was a constant instead, then computing the path is unnecessary since $getSCEV$ would return a $C$ object (regardless of the context). When computing the SCEV for the operand, we keep track of the old call string $CS_{old}$, set the current call string ($CS$) to $CS_{new}$, compute the SCEV for the operand, and then reset the call string to $CS_{old}$. In Algorithm 6.4, we update
the call string by first removing entries from it and then extend it by calling procedure
addCallSitesToCS, presented in Algorithm 6.5.

**Algorithm 6.4** Procedure *updateCallString*. Used to update the call string *CS* (remove and add call sites), with the path to a temporary *Op*. Recall that procedure *ProgPointToInst* is described in Section 2.3. It accepts a program point as input and returns the instruction whose value it holds.

1: proc updateCallString(*I* : *INST*, *Op* : *TMP*) begin
2: \[ PP := \text{getPropagationPoint}(*I*, *Op*) \{ \text{see Algorithm 5.3} \} \]
3: if *PP* is not the entry to *I*'s parent procedure then
4: addCallSitesToCS(*I*, *Op*)
5: return
6: while |*CS*| > 0 do
7: \[ cs := \text{pop}(*CS*) \]
8: \[ PP := \text{getPropagationPoint}(*\text{ProgPointToInst}(cs)*, *Op*) \]
9: if \( \text{ProgPointToInst}(PP) = \text{call...} \) then \{ *PP* is not the entry to a procedure\}
10: addCallSitesToCS(*\text{ProgPointToInst}(PP)*, *Op*)
11: return
12: end

Algorithm 6.4 updates the current call string when visiting the operand *Op* of instruction *I*. When the propagation point of *Op* at *I* is a call site, procedure *addCallSitesToCS* is used to add additional call sites to the end of *CS*. Otherwise, if the propagation point of *Op* is the entry to procedure *P*, we pop from *CS* until it contains only the call sites on a path to *Op*. This occurs when the propagation point of *Op* at *cs* (the call site we just popped on line 7 in Algorithm 6.4) is another call site rather than the entry to the procedure in which *cs* is located. Once *cs* is found, we trace the call graph path to *Op* by calling procedure *addCallSitesToCS*.

Algorithm 6.5 presents procedure *addCallSitesToCS*, which computes the sequence of call sites that makes up the call graph path on which a temporary *Op* is propagated to instruction *I*. Note that if the propagation point of *Op* is the entry of the parent procedure of *I*, then procedure *updateCallString* will not invoke procedure *addCallSitesToCS*. Starting with instruction *I*, we traverse the IR in reverse-dominator order until a call site reaching procedure *Q* is found. Instructions in a basic block *BB* are visited in reverse order until the entry instruction is reached. At that point, we begin iterating over the
Algorithm 6.5 Procedure addCallSitesToCS. It Extends the call string CS with the call graph path between instruction $I$ in procedure $P$ and one of its operands, a temporary $Op$ that is defined in procedure $Q \neq P$. In this algorithm, $\text{ImmediateDomInst}(ci)$ returns the previous instruction if it exists (i.e. not at the start of the basic block) or the terminator instruction of the immediate dominator of $ci$’s parent otherwise. In our implementation, the efficiency of the algorithm is improved by iterating over a dominator tree composed of only call sites.

**Require:** Propagation point of $Op$ to $I$ is not the entry to $I$’s parent procedure ($P$)

1: proc addCallSitesToCS($I : \text{INST}, Op : \text{TMP}$) begin
2:   $ci := I$
3:   if $I = \phi \ldots , \langle \text{Pred}, Op \rangle , \ldots$ then
4:      $ci := \text{getTerminator}(\text{Pred})$
5:   Q := getParentProcedure(TempToInst($Op$))
6:   while $ci \neq 0$ do
7:      if $Q \in \mathcal{RPC}[ci]$ then
8:         $cs := \text{InstToProgPoint}(ci)$
9:         push($CS, cs$)
10:        callee := Targ($ci$)
11:        if callee $\neq Q$ then
12:           addCallSitesToCS(getEndInst(callee), $Op$)
13:      return
14:   $ci := \text{ImmediateDomInst}(ci)$
15: end
instructions in the immediate dominator of $BB$. If a call instruction $ci$ can reach procedure $Q$, we add its call site ($cs$) to the end of the call string. If $ci$ targets procedure $Q$, then we return from the algorithm. Otherwise, we call $addCallSitesToCS$ to derive the call graph path from the target of $ci$ (starting from the return statement) to procedure $Q$.

To simplify the presentation of procedure $addCallSitesToCS$, we omitted details for handling recursive cycles. Recursive cycles occur when a call site $cs \in CS$ is already located in the procedure we target (i.e. callee). When this condition is encountered we stop tracing the recurrence relation across procedures.

**6.5.4 File-Position Induction Variable Analysis**

In order for us to analyze the evolution of the file position, we make a number of assumptions. First, we assume that the program is sequential and that it does not share file identifiers with another process (e.g. through forking or $execv$). Second, in order to analyze the file position, we need to account for the impact of each external call. Hence, we assume that all external calls that can change the file position as well as their actual impact are known at compile time. Moreover, in order to apply the file-position induction variable analysis towards finding trip counts and parallelization, we must make sure that files are not changed while running the program.

In order to compute the file-position evolution within loops, we modify the pointer analysis and ISSA construction. The pointer analysis is changed to track the file objects and identifiers that are referenced at file operations. Prior to ISSA construction, we create additional SSA variables, which will track the file position. Then, we apply an IR traversal that will identify call instructions that modify the file position and update the newly created variables accordingly. When ISSA form is generated, each use of the newly created SSA variables is replaced with a single definition and the induction variable analysis can be leveraged to compute their evolution.
Pointer analysis changes

We modified the pointer analysis to identify the files associated with temporaries. First, a file object is created at each call instruction $ci$ that targets either procedure $open$ or $fopen$ and the return value (for $ci$) is set to the file object. The impact of a call instruction that is passed a file-stream pointer $pn$ and targets procedure $freopen$ is modeled by first creating $pnew$, a new file object. Then, we add $pnew$ to the point-to set of both $pn$ and the return value of the call instruction invoking $freopen$. Both file streams and file identifiers are associated with file objects. Conversions between file streams and file identifiers via calls to $fdopen$ and $fileno$ are modeled as copy instructions. We let the return value of the call instruction (that calls $fdopen$ or $fileno$) point to the value of the first parameter. Lastly, $stdin$, $stdout$, and $stderr$ are assigned their own file object as well.

SSA variables tracking the file-position

Once the pointer analysis completes, we leverage the ISSA construction algorithm to compute the evolution. For each singleton file object, we create a file stream variable or a file identifier variable, which are global variables that are constrained to the size of the file they represent. File variables are initialized to 0 after their corresponding file object is created and are incremented, reset to 0, or set to an unknown value to model file operations. When both a file stream and file identifier reference the same file object, we create both file stream and file identifier variables and initialize them to 0.

Modifications to capture the impact of external calls on the file-position

During ISSA generation, we traverse the IR and capture the impact of each file read, write, and file-position setting operation (on the file position) by updating the corresponding file variables. The file variables are updated after each call instruction $I$ that targets a file read, write, or file-position setting function. First, let us assume that only a single file object, which is associated with the file variable $gv$, can be referenced at
I. If $I$ is a call to `getc`, `putc`, `getchar`, `putchar`, `fgetc`, or `fputc`, then we increment $gv$ by 1. If $I$ is a call to a different file read or write function, then we increment $gv$ by $\mathcal{NN}$. Recall that $\mathcal{NN}$ is a non-negative constant discussed in Section 6.4.2. Finally, we handle file-position setting functions such as `fseek` and `lseek`, when they reset the position to the start of the file by assigning 0 to $gv$. Otherwise, $gv$ is assigned an undefined constant $Unkn$ (the unknown value).

While incrementing the file position by 1 can advance $gv$ past the end of the file, recall that we constrain $gv$ to the size of the file that it represents. In other words, when computing the trip count or dependencies, the file position will be equal to $\min(gv, \text{file.size})$. Since operations that decrement the file position are handled by setting $gv$ to $Unkn$, $gv$ correctly models the file position when it goes past the end of the file.

When we read and write data to a file stream, the file position is unknown because the file contents are buffered. This does not invalidate our induction variable analysis as continuous reads and writes to the file stream advance the file position forward in a predictable manner. However, file-identifier reads and writes are not buffered. Due to this difference, interchanging between file stream and identifier reads and writes increases the file position in an unpredictable manner. In order to model this behavior, we update both the file stream variable and file identifier variable at each operation. Let us assume $fs$ is a file stream variable and $fi$ is a file identifier variable corresponding to a file object $F$. When encountering any file stream reads or writes to $F$ we set $fi := fi + \mathcal{NN}$ and for file identifier reads or writes we set $fs := fs + \mathcal{NN}$. When a `fflush` call is made we set $fi := fs$ and all other file position setting routines update both $fi$ and $fs$.

Note that an operand of a call instruction that modifies the file-position may be aliased to more than one file stream or file identifier. In this scenario, we need to conditionally update the file-position of more than one file object. In order to do this, a sequence of $\phi^S$ instructions is inserted to conditionally assign the new value to the file variables that correspond to the file objects that can be referenced. The pointer value of each $\phi^S$
instruction is \( pn \), which is the file stream or file identifier used at the call instruction. A \( \phi^S \) instruction is inserted for each pointed to file object. The \( \phi^S \) instruction compares \( pn \) to the value returned from the \textit{open}, \textit{fopen}, \textit{freopen}, or \textit{fdopen} call instruction that the file object corresponds to. In order to compare file identifiers, we extended the semantics of the \( \phi^S \) instruction to compare integers.

### Leveraging infrastructure to compute the file-position

Since all file variables are SSA variables, when ISSA form is generated each use of a file variable is replaced with a single definition. In order to improve precision, we fold addition, \( \phi^V \), \( \phi^C \), and \( \phi \) instructions involving the constants \textit{Unkn} and \textit{NN}.

In the ISSA form, \( \phi \) instructions corresponding to file variables are inserted into the loop header. The evolution of the file position is computed by calling procedure \textit{getSCEV}, described in Algorithm 6.1, and passing it a \( \phi \) instruction that is located in the loop header and corresponds to a file variable \( \text{var} \). The returned SCEV (which can be a linear, polynomial, monotonic, or predicated induction variable) represents the evolution of the file position for the corresponding file stream or file identifier of \( \text{var} \).

### Example 6.3 File-Position Analysis for the C source code in Figure 6.2(a)

Consider the IR shown in Figure 6.2(a). The result of the pointer analysis relevant to us is that \( pn \) in procedure main points-to only the file-object created on line 14. Using this result, we will transform the IR as shown in Figure 6.3(a). First, we create a file stream variable (\( fs \)) to track the file position of the file pointed to by \( pn \). On line 17 in Figure 6.3(a) we initialize the file-position that \( fs \) corresponds to by assigning it 0. Since the \textit{getc} call increments the file-position by 1, we increment \( fs \) by 1 immediately afterwards, on line 8.

With these changes, the ISSA form shown in Figure 6.3(b) is generated rather than the ISSA form in Figure 6.2(b). The difference between these programs is that in Fig-
(a) Modifications in the source code from Figure 6.2(a), as described in Section 6.5.4, to derive the file-position evolution.

(b) The ISSA form for Figure 6.3(a). It is identical to the ISSA form Figure 6.2(b) when excluding the bold-font expressions.

Figure 6.3: Example based on the C source code in Figure 6.2 that illustrates the file-position analysis.
ure 6.3(b) we insert three new instructions. Due to the new assignments to variable $fs$ a $\phi^V$ instruction assigned to $\%fs0$ is inserted at the entry to procedure Next on line 3. In addition, a $\phi$ instruction assigned to $\%fs1$ is inserted in the loop header within procedure main on line 22. Its incoming value from the latch basic block is $\%fs2$ (equals $\%fs0 + 1$), which is defined in procedure Next on line 5 in Figure 6.3(b). For the $\phi$ instruction held in the temporary $\%fs1$, we determined the entry value is equal to 1 (rather than a $\phi^C$ instruction) using the one-level context-sensitive constant propagation.

Using our induction variable analysis, we determine the evolution of $\%fs1$, and classify it as a linear induction variable with base 1 and increment 1.

6.6 Case Studies

In this section, we present a number of examples from benchmarks in MediaBench [31] and SPECINT2000 [1] where additional induction variables are identified. In Section 6.6.1, we discuss a case study where a global variable is used as a loop index. Section 6.6.2 contains a scenario where a linear induction variable is identified by computing the interprocedural recurrence relation. In Section 6.6.3, we show a section of code where the induction variable analysis determines that a structure field allocated on the heap is a linear induction variable. In Section 6.6.4, we present an example where the file-position evolution is computed and discuss how this result can be applied towards computing the loop trip count.

6.6.1 ScalarGlobals

In Figure 6.4(a), we show a loop ($L$) that is taken from the benchmark 300.twolf in SPECINT2000 [1]. Note that global variable $row$ is a loop index, assigned 1 at the start of the loop and incremented by 1 after every iteration. Since global variables $row$ and $numRows$ are SSA variables, uses of these variables are replaced with the corresponding
Chapter 6. ISSA-Based Interprocedural Induction Variable Analysis

config1() {
    ... 1
    for (row = 1; 2 row <= numRows; 3 row++) {
        rowArray[row].endx1 = -1; 4 rowArray[row].startx2 = -1; 5 }
    ... 6
}

(a) C source code.

Figure 6.4: C source code fragment illustrating a loop in the benchmark 300.twolf in SPECINT2000 [1] where a global variable (row) is a linear induction variable.

(b) Relevant instructions in the ISSA form for Figure 6.4(a.)

\[
\text{config1}() \{ \begin{array}{l}
\text{BB0:} \quad \text{br BB2;} \\
\text{BB1:} \quad \text{br BB2;} \\
\text{BB2:} \quad \%\text{row0} := \phi (\text{BB0},#1),(\text{BB1},\%\text{row1}); \\
\quad \%\text{v0} := \text{leq} \%\text{row0},\%\text{numRows0}; \\
\quad \text{br} \%\text{v0}, \text{BB1},\ldots; \\
\quad \ldots \\
\end{array} \}
\]

Hence, the \(\phi\) instruction \%\text{row0} is inserted on line 9 in Figure 6.4(b). When applying the procedure \text{getSCEV} on \%\text{row0}, the returned SCEV is \(CR\{L,C\langle 1 \rangle,C\langle 1 \rangle\}\) indicating that \%\text{row0} (that corresponds to the global variable \text{row}) is a linear induction variable with base 1 and increment 1. The SCEV is derived by computing the recurrence relation, which is equal to \%\text{row0}_{\text{nextiter}} = \%\text{row0}_{\text{curriter}} + 1.

In the ISSA form, the reference to \text{numRows} is replaced with \text{numRows0}, a loop invariant temporary, which is defined outside the procedure \text{config1}. Determining that \%\text{row0} is a linear induction variable enables us to evaluate the trip count, which is equal to \text{numRows0}. Furthermore, because we determined that \%\text{row} is a linear induction variable, a dependency test can conclude that the loop can be parallelized.
void build_deletable(void) {
    for (i=0; i<N_words; i++) {
        deletable[i] = (char *) CI1: xalloc(N_words+1);
    }
    . . .
}

void * xalloc(int size) {
    %size0 := φV . . . , ⟨CI1,%v0⟩ . . . ;
    %siu2 := φV . . . , ⟨CI1,%siu0⟩ . . . ;
    %v1 := add %size0,%siu2;
}
%v1, we push CI1 on top of the call string and invoke getSCEV with the argument %v1. Since %v1 adds temporaries (%siu2 and %size0) holding the value of φV instructions, we evaluate %siu2 and %size0 using the call string. The SCEVs for the temporaries %siu2 and %size0 will be equal to I⟨%siu0⟩ and +⟨I⟨%nwords0⟩,C⟨1⟩⟩, respectively. Note that +⟨I⟨%nwords0⟩,C⟨1⟩⟩ is loop invariant, since %nwords0 is defined outside the loop. Using this result, we will determine that the recurrence relation of %siu0 in this loop is %siu0_{nextiter} = %siu0_{curriter} + %nwords + 1. This expression is then classified as a linear induction variable with base %outv (the incoming value of %siu0 from the loop entry, which is loop invariant) and an increment of %nwords + 1.

### 6.6.3 Heap-Allocated Structure Fields

![C source code](image)

(a) C source code.

![Relevant instructions in the ISSA form](image)

(b) Relevant instructions in the ISSA form for Figure 6.6(a).

Figure 6.6: Example illustrating a heap-allocated induction variable in the benchmark JPEG in MediaBench [31] (Relevant source code can be found in the files jcmaster.c and jcmaint.c).

In Figure 6.6(a), we can compute the evolution of a heap-allocated structure field in the benchmark JPEG, which is part of MediaBench [31]. Figure 6.6(a) contains relevant
sections of code while Figure 6.6(b) provides the corresponding ISSA form.

The expression main→cur.jMCU.row corresponds to a structure field, which we refer to as cur, that is allocated on the heap in procedure jinit.c.main.controller (file jc-mainct.c). Because the allocation site is a singleton, cur is an SSA variable. Note that cur is incremented by 1 on line 9 in Figure 6.6(a). Aside from this store instruction, no other values are assigned to cur within the while loop on lines 7–10. In the ISSA form, \%v0 corresponds to the value of cur when entering the procedure process_data_simple_main and as such, it is loop invariant (in relation to the while loop on lines 7–10).

Hence, in the ISSA form, the \( \phi \) instruction defining \%cur0 on line 11 in Figure 6.6(b) is inserted. When we apply the induction variable analysis, \%cur0 is classified as a linear induction variable whose base is \%v0 and its increment is 1.

Note that we can constrain the trip count of the loop on lines 7–10 in Figure 6.6(a). One of the reasons this can be done is because we determined that \%cur0 is a linear induction variable. Another reason is that ISSA form enables us to determine that the value of cinfo→total.jMCU.rows on line 7 in Figure 6.6(a) is loop invariant (\%total0), since it is passed into procedure process_data_simple_main. Because of this, the trip count of the loop on lines 7–10 in Figure 6.6(a) can be constrained to \( \max(0, \%total0 - \%v0) \).

### 6.6.4 File-Position Induction Variables

On line 4 in Figure 6.7, we show a file-read operation involving the file stream pointed-to by in. The variable in is either equal to stdin or points-to a file stream fd. As we can see, the file-position is incremented by a value between 0 and 33 (size of s) on each iteration of the loop. Hence, the file-position of the file stream pointed-to by in is monotonically nondecreasing.

This is captured by our file-position induction variable analysis using two \( \phi^S \) nodes that conditionally increment the file stream variables for stdin (stdin\( _V \)) and fd (fd\( _V \)) with a \( \mathcal{N} \mathcal{N} \) object. When computing the SCEV for each temporary that holds the value of
a $\phi$ instruction which is inserted in the loop header (corresponding to $stdin_\nu$ and $fd_\nu$),
our induction variable analysis computes a predicated SCEV $PredS$ that is reduced to
a monotonically non-decreasing induction variable.

Note that we can refine this result; by noting that if $cc$ is not equal to 33, the loop
exits through the branch that is taken on line 5 in Figure 6.7. ISSA form helps us make
this refinement, as it takes $O(1)$ time to identify all uses of a definition to an SSA variable;
allowing us to constrain values based on the usage point. With this refinement, the file
position of the file stream pointed-to by $in$ is classified as a (predicated) linear induction
variable with base 0 and an increment of 33.

Moreover, if the file-stream pointed-to by $in$ is not associated with the terminal and
its corresponding file is not modified during the loop then the trip count is loop invariant.
Using the system calls `stat` or `fstat` the file size (`sz`) can be obtained and used to compute the trip count, which is equal to $\lceil \frac{sz}{33} \rceil$.

### 6.7 Experimental Evaluation

<table>
<thead>
<tr>
<th>MediaBench</th>
<th>SPECINT2000</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Time (ms)</td>
<td>Name</td>
</tr>
<tr>
<td>G721</td>
<td>10</td>
<td>164.gzip</td>
</tr>
<tr>
<td>GSM</td>
<td>43</td>
<td>175.vpr</td>
</tr>
<tr>
<td>JPEG</td>
<td>215</td>
<td>181.mcf</td>
</tr>
<tr>
<td>MPEG2</td>
<td>385</td>
<td>186.crafty</td>
</tr>
<tr>
<td></td>
<td></td>
<td>197.parser</td>
</tr>
<tr>
<td></td>
<td></td>
<td>254.gap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256.bzip2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>300.twolf</td>
</tr>
</tbody>
</table>

Table 6.1: The time it takes to perform the induction variable analysis in millisecond.

We evaluated the induction variable analysis by counting the number of induction variables and loops with constant or invariant trip count. Our comparison was with the LLVM induction variable analysis, which relies on SSA form. The number of additional induction variables identified is a direct metric for the performance of our algorithm. Through the induction variable analysis and ISSA form, we can compute the trip count of various loops. Identifying the trip count is highly important when testing for dependency, parallelizing loops, and various optimizations including loop unrolling and software pipelining.

Our evaluation was on a set of MediaBench [31] and SPECINT2000 [1] benchmarks as well as the uIP benchmark [21], which is a TCP/IP stack designed for tiny embedded systems. The experimental setup is described in detail in Section 4.8 and Section 4.8.1 in Chapter 4. The runtime of the induction variable analysis was, as shown in Table 6.1, at most, two and a half seconds.

In some benchmarks, such as `uIP`, global variables are used as loop indices. While the
### Table 6.2: The number of induction variables found. Columns labeled \textit{LLVM} contain the baseline numbers. We differentiate between induction variables found by tracing the recurrence relation interprocedurally in the columns labeled \textit{Inter} (with) and \textit{Intra} (without).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Linear Induction Variables</th>
<th></th>
<th>Monotonic Induction Variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\textit{LLVM}</td>
<td>\textit{Intra}</td>
<td>(\Delta) (%)</td>
<td>\textit{Inter}</td>
</tr>
<tr>
<td>GSM</td>
<td>52</td>
<td>52</td>
<td>0.0%</td>
<td>52</td>
</tr>
<tr>
<td>G721</td>
<td>8</td>
<td>9</td>
<td>12.5%</td>
<td>9</td>
</tr>
<tr>
<td>JPEG</td>
<td>274</td>
<td>279</td>
<td>1.8%</td>
<td>279</td>
</tr>
<tr>
<td>MPEG2</td>
<td>69</td>
<td>70</td>
<td>1.4%</td>
<td>70</td>
</tr>
<tr>
<td>164.gzip</td>
<td>92</td>
<td>100</td>
<td>8.7%</td>
<td>100</td>
</tr>
<tr>
<td>175.vpr</td>
<td>225</td>
<td>228</td>
<td>1.3%</td>
<td>229</td>
</tr>
<tr>
<td>181.mcf</td>
<td>7</td>
<td>8</td>
<td>14.3%</td>
<td>8</td>
</tr>
<tr>
<td>186.crafty</td>
<td>135</td>
<td>142</td>
<td>5.2%</td>
<td>143</td>
</tr>
<tr>
<td>197.parser</td>
<td>161</td>
<td>170</td>
<td>5.6%</td>
<td>189</td>
</tr>
<tr>
<td>254.gap</td>
<td>867</td>
<td>874</td>
<td>0.1%</td>
<td>874</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>73</td>
<td>80</td>
<td>9.6%</td>
<td>80</td>
</tr>
<tr>
<td>300.twolf</td>
<td>275</td>
<td>314</td>
<td>14.2%</td>
<td>314</td>
</tr>
<tr>
<td>uIP</td>
<td>10</td>
<td>20</td>
<td>100.0%</td>
<td>20</td>
</tr>
<tr>
<td>Average</td>
<td>13.3%</td>
<td>14.4%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The benefit of our proposed ISSA-based induction variable analysis varies with benchmarks, we believe it greatly improves on previous work in such scenarios. Hence, unlike previous chapters, we study the benefit of our ISSA-based induction variable analysis for the benchmark \textit{uIP} was well.

As it can be observed from Table 6.2, using our algorithm, we identified more linear and monotonic induction variables than the LLVM infrastructure. The largest absolute improvement was observed in the benchmark \textit{300.twolf} because of the frequent use of global variables as loop indices (mostly in the procedures \textit{config1} and \textit{configure}). For the same reason, the largest relative improvement was observed in the benchmark \textit{uIP} where a single global variable was used as the loop index for a number of loops. In these benchmarks, the global variables are used as array indices, hence our induction variable analysis allows us to accurately compute the loop carried dependency and to
Table 6.3: The number of loop trip counts computed. Columns labeled LLVM contain the baseline numbers. The ISSA column contains the number of trip counts found due to ISSA form alone, while the ISSA+IV column also considers the newly discovered induction variables when computing the trip count.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Constant Trip Counts</th>
<th>Loop Invariant Trip Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LLVM</td>
<td>ISSA</td>
</tr>
<tr>
<td>GSM</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>G721</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>JPEG</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>MPEG2</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>164.gzip</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>175.vpr</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>186.crafty</td>
<td>78</td>
<td>78</td>
</tr>
<tr>
<td>197.parser</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>254.gap</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>300.twolf</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>uIP</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Average</td>
<td>1.02</td>
<td>1.1</td>
</tr>
</tbody>
</table>

parallelize a number of loops. After the benchmark 300.twolf, the second largest absolute improvement was observed in the benchmark 197.parser, which profited heavily from context sensitivity. Of the additional 28 linear induction variables that were identified over the LLVM infrastructure, 19 were discovered through tracing the recurrence relation interprocedurally in a context-sensitive manner. While in other benchmarks we found fewer linear induction variables, some were still quite useful. For instance, as shown in Section 6.6.3, we can use this new information to constrain the trip count.

The relative increase in the number of monotonic induction variables that were identified was 49.1%, which is much higher than the relative increase in the number of linear induction variables. This result supplements the work done by Gerlek et al. [25], which observed that few (less than 2%) monotonic induction variables can be identified on SSA form. We also noted that many of the newly identified monotonic induction vari-
variables were incremented under loop invariant conditions and could be resolved to linear induction variables when creating multiple versions of certain sections of code.

We were able to compute a large number of trip counts, by identifying additional induction variables and through ISSA form. In the benchmark \textit{uIP}, we leveraged the results of the induction variable analysis, which identified 10 more induction variables, to compute a constant trip count for twice as many loops. Similarly, in the benchmark \textit{300.twolf}, we increased the number of loops with invariant trip count by over 7X. This was accomplished by identifying more linear induction variables as well as using the ISSA form to identify loop invariant values.

In other benchmarks, as shown in Table 6.3 (under the column labelled \textit{ISSA+IV}), we also observed a large increase in the number of computable loop invariant trip counts. As it can be seen (from the difference between the \textit{ISSA+IV} and \textit{ISSA} labeled columns), the increase in the number of computable loop invariant trip counts is largely attributed to ISSA form that enables the scalar evolution pass to quickly determine that a value is loop invariant. A MOD/REF analysis can also be used to identify loop invariant values or hoist them outside the loop. However, we need to identify the program variables referenced using a point-to graph and the scalar evolution pass must search the call sites of the loop to determine whether a variable is loop invariant.

In Table 6.4, we present the results of our file-position induction variable analysis. In the benchmark \textit{JPEG} from the MediaBench suite, we identified the highest number of file-position induction variables. For a very large number of the file-position induction variables the recurrence relation was interprocedural. In fact, as shown in the corresponding \textit{Inter} column, the file-position induction variable analysis profited the most from tracing the recurrence relation interprocedurally. Moreover, the file-position induction variable analysis was very consistent, since a few induction variables were identified in almost every benchmark.
Chapter 6. ISSA-Based Interprocedural Induction Variable Analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Linear Intra</th>
<th>Linear Inter</th>
<th>Monotonic Intra</th>
<th>Monotonic Inter</th>
<th>Predicated Intra</th>
<th>Predicated Inter</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSM</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>G721</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JPEG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>MPEG2</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>186.crafty</td>
<td>0</td>
<td>0</td>
<td>28</td>
<td>29</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>197.parser</td>
<td>4</td>
<td>4</td>
<td>15</td>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>254.gap</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>uIP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.4: Performance of the file-position induction variable analysis. We differentiate between induction variables found when tracing the recurrence relation interprocedurally in the columns labeled $\text{Inter}$ (with) and $\text{Intra}$ (without).

6.8 Summary

In this chapter, we demonstrated the benefit of a context-sensitive interprocedural induction variable analysis that computes the evolution of global variables, singleton heap locations, structure field, and the file-position.

In the future, we would like to improve the file-position induction variable analysis to interpret branches and capture their impact. This would allow us to compute trip counts as well as improve the precision of our analysis. We believe that determining the precise evolution of the file-position can be used to specialize program segments based on the input to the program as well as parallelize loops and procedures.
Chapter 7

Conclusions and Future Work

In contrast to SSA, both the set of SSA variables and the scope of values are extended in ISSA. While seemingly a natural extension, the tradeoff between the benefit and cost of ISSA form was never thoroughly evaluated in the literature. In this dissertation, we investigated the integration of our ISSA form into a compiler and evaluated the impact on various compiler optimizations. In this study, we have shown that ISSA form can be efficiently constructed for a large number benchmarks. Moreover, we proposed an algorithm that performs out-of-ISSA translation quickly, without degrading the performance of the code. Furthermore, we demonstrated that ISSA form can be seamlessly leveraged by many compiler optimizations to obtain a benefit. Given these observations, we believe that our research forms a solid foundation, upon which future work can build.

7.1 Constructing ISSA Form

In Chapters 3 and 4, we described the ISSA form and provided an algorithm to construct it. In contrast to previous work, we construct ISSA form IR rather than represent ISSA in a separate data structure. The construction of ISSA form took less than 10 seconds on most benchmarks, with the exception of the benchmarks 197.parser, 254.gap, and 300.twolf in SPECINT2000 [1], whose ISSA form was constructed in 21.52 seconds, 91.17
seconds, and 38.63 seconds, respectively. Moreover, we extend the scope of values to the whole program, which enables us to fold $\phi^V$ and $\phi^C$ instructions. We observed that our proposed copy propagation algorithm reduced the number of $\phi^V$ and $\phi^C$ instruction by 44.5%, on average. In addition to these contributions, we showed that a field-sensitive pointer analysis reduces the size of the input and output sets by a factor of 12.2 (i.e. $\text{REF}$ and $\text{MOD}$ in Section 4.5). The ISSA form described as well as the construction algorithm have been published in [13].

When examining the IR, we noted that $\phi$ instructions accounted for over 50% of all newly inserted instructions during ISSA construction and consumed over 54% of the space. In fact, ISSA could not be constructed for the benchmarks 255.vortex and 176.gcc because of the memory consumptions, which is mostly attributed to $\phi$ and $\phi^V$ instructions.

### 7.2 Out-of-ISSA Translation

In Chapter 5, we presented an out-of-ISSA translation algorithm which enables us to convert back to SSA form without degrading performance. The out-of-ISSA translation was much faster than the ISSA form construction (at most three seconds on all the benchmarks). Hence, we are confident that transforming the IR back to SSA form is always feasible using our approach. Moreover, while a straightforward extension of out-of-SSA translation algorithms degrades performance by a factor of 1.5 in comparison to the LLVM infrastructure, our proposed algorithm actually improves performance by a factor of 1.02.

Through small modifications, we adapted a number of LLVM passes to ISSA form and quantified the benefit over the LLVM infrastructure. The constant propagation pass folded 10.8% more instructions. On average, the common subexpression elimination pass removed 42.9% more instructions and due to dead code elimination and constant
propagation the resulting IR had 1.8% fewer basic blocks. In addition to these optimizations, by applying the out-of-ISSA translation certain parameters, load instructions, and store instructions were no longer required. This enabled us to remove 5.0% more store instructions, 23.1% more load instructions, and 14.2% more parameters.

7.3 ISSA-Based Interprocedural Induction Variable Analysis

In Chapter 6, we leveraged the ISSA form to extend the induction variable analysis interprocedurally. This required only 370 lines of C++ code, which were used to handle new instructions in ISSA form as well as interprocedural references. The interprocedural induction variable analysis identified 14.4% more polynomial induction variables and 49.1% more monotonic induction variables than the LLVM baseline. Using ISSA form and the newly identified induction variables we were able to compute 1.1 times more constant trip counts and 2.6 times more loop invariant trip counts. Moreover, we present an algorithm that computes the file-position evolution by leveraging the induction variable analysis and ISSA construction. We quantify the impact of our proposed approach and note that we can identify many file-position induction variables. When summing our results, 8% of all induction variables would represent the file-position. Our work on the interprocedural ISSA-based induction variable analysis has been published in [14].

7.4 Limitations

The ISSA construction algorithm that was presented in this dissertation has a three major limitations. First, it does not scale to large benchmarks (in terms of lines of code) such as 176.gcc and 255.vortex in SPECINT2000 because the system we used ran out of memory. In addition, the construction of ISSA form takes longer than many compiler
passes, including the SSA form construction. Second, the set of SSA variables does not include arrays and most heap variables. Including such program variables in the set of SSA variables can ameliorate the benefits derived from ISSA. Third, our implementation does not support various object-oriented features which limits us to analyzing benchmarks written in C (and not C++).

The out-of-ISSA translation also has a number of limitations. First, while it is much faster than the ISSA form construction, it is slower than many compiler passes. Moreover, constructing and translating out-of-ISSA multiple times is very costly with the current approach. Second, the out-of-ISSA translation only uses global variables and existing parameters to propagate values across procedures. Performance may be further improved by using additional variables to propagate values across procedures.

Of these limitations, we believe that the high cost of constructing ISSA and the fact it does not scale to large benchmarks is a fundamental problem. The primary reason this happens is that the impact of a definition is extended to the whole program. Note that this is a key feature in ISSA. In order to adapt ISSA into a production compiler this limitation needs to be addressed and in Section 7.5.1 we propose future work to tackle this problem.

7.5 Future Work

There are several ISSA-related research directions. Below, we discuss extensions of this work that focus on integrating ISSA form in compilers and applying it towards various applications.

7.5.1 Reducing the Cost of ISSA Construction

In Chapter 4, we explored the use of a number of techniques to reduce the memory space consumed by ISSA form. However, our construction algorithm did not scale to
the benchmarks 176.gcc and 255.vortex in SPECINT2000 [1] due to the space consumed by φ instructions. In order to integrate ISSA form into a compiler, we need to further reduce its construction time and memory consumption. We can address these challenges in a number of ways.

Future work can limit the ISSA construction to a set of SSA variables and a set of procedures. The selection of SSA variables and procedures can be done by leveraging heuristics, which attempt to maximize the benefit of ISSA while reducing its space consumption and construction time. Another optimization is to devise an algorithm that predicts the impact of the selected procedures and SSA variables on the construction time and space consumption. Using such an algorithm, we can constrain the cost of constructing ISSA, which would enable us to leverage ISSA form in a production compiler.

Space consumption can also be reduced by using a more efficient ISSA representation. For instance, the operands of each φ instruction include both the incoming values as well as symbolic labels (for basic blocks). However, each φ instruction in the same basic block has an identical set of predecessors. As such, we can impose an order for the incoming values and use the location of an incoming value to retrieve its corresponding basic block. In such a scenario, we only need to keep a reference to the incoming value, which would enable us to cut the number of φ instruction operands in half. Another research direction is to represent the IR using more space efficient data structures, such as the Binary Decision Diagram (BDD). Finally, we can reduce the space consumption by making changes to the proposed ISSA. For instance, we can collapse φ instructions that do not offer much benefit into a single node, as was proposed by Chow et al. [16].

7.5.2 Propagating Values Across Procedures

In our work, we only explored the use of global variables and existing procedure parameters to propagate values across procedures. While this enabled us to integrate ISSA form into a compiler and assess the benefit, converting stack and heap allocated variables to
globals may have a number of drawbacks. We can improve on our out-of-ISSA translation algorithm by propagating values across procedures in both stack and heap allocated variables. Moreover, we can propagate values across procedures through newly introduced parameters, or avoid propagating certain values altogether, by applying inlining.

### 7.5.3 Applications of ISSA

In this dissertation, we demonstrated the benefit of ISSA form to a number of compiler analyses and optimizations. Nevertheless, there are many other compiler passes that can be extended to profit from ISSA and it would be interesting to evaluate the benefit. Beyond existing compiler passes, there are a number of applications that can benefit from ISSA form and we discuss a few.

First, ISSA can be leveraged for program specialization. Using program specialization, we can fold branch and arithmetic instructions, remove unreachable code, and identify more parallelism. Therefore, program specialization can improve the runtime of a program under specific inputs. By leveraging ISSA form, we can approximate the impact of specializing a section of code for a given temporary quickly. Beyond being used to identify temporaries to specialize, we can determine the instructions impacted by this change, which enables us to apply the transformation as well. As such, we believe that ISSA form can be leveraged to simplify program specialization and improve the derived benefit.

Second, we can leverage ISSA to perform value inference. At conditional branch instructions, we can infer the value of operands based on the basic block taken. Furthermore, if a given procedure $P$ is called at an indirect call instruction $ci$, then we may infer that the pointer value of $ci$ is equal to $P$ (at the entry to $P$). Investigating the impact of ISSA form on value inference is a promising research direction, in part due to the explicit identification of the program-wide uses of a temporary in ISSA form.

Third, we can build upon the file-position analysis by handling file operations more
precisely as well as using value inference and program specialization to obtain more linear evolutions (rather than other evolution kinds such as monotonic and predicated). Moreover, future research can leverage the file-position analysis to compute loop-carried dependencies and parallelize loops.

7.6 Closing Remarks

In this dissertation, we presented techniques to integrate ISSA form into a compiler and demonstrated a benefit to a number of compiler analyses and optimizations. In most compiler passes leveraging ISSA form, we observed a substantial improvement while making only minor modifications to the code.

This section builds on this work and proposes new approaches to improve the ISSA form construction, out-of-ISSA translation, and discusses optimization opportunities enabled by ISSA. We are optimistic that the techniques and future research directions presented in this dissertation pave a path towards adapting ISSA form into compilers and leveraging it to simplify and improve interprocedural analyses and optimizations.
Bibliography


[38] Sebastian Pop, Albert Cohen, and Georges-André Silber. Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and*


