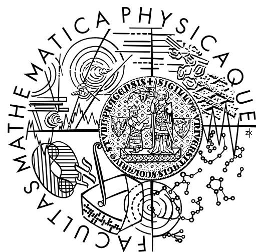

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Filip Kastl

An alternative SSA construction algorithm for GCC

Department of Applied Mathematics

Supervisor of the bachelor thesis:	doc. Mgr. Jan Hubička, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Dedication. I would like to thank Jan Hubička for putting much of his time into guiding me. I would also like to thank my close ones for cheering me on even when I felt overwhelmed.

Title: An alternative SSA construction algorithm for GCC

Author: Filip Kastl

Department: Department of Applied Mathematics

Supervisor: doc. Mgr. Jan Hubička, Ph.D., Department of Applied Mathematics

Abstract:

SSA form is a very important concept in compiler internal code representation. Φ-functions are an integral part of SSA form. Braun, Buchwald, Hack, Leißa, Mallon and Zwinkau introduce a new algorithm for SSA construction and another related algorithm for reducing the number of Φ-functions. These algorithms are not yet implemented in the GCC compiler.

Firstly, we introduce, implement and test a basic code generation API based on the SSA construction algorithm. We list the possible extensions and usecases of the API. Then we implement the Φ optimization as a standalone pass. We use it to measure the number of redundant Φ-functions produced by other GCC passes. Finally, we conclude that GCC would benefit from including both of these algorithms.

Keywords: compiler, SSA form, optimization

Contents

Introduction **3**

1 Background **5**

1.1 GIMPLE intermediate language 5
1.2 Optimization passes 6
1.3 Control Flow Graph 6
1.4 Static Single-Assignment Form 7
1.4.1 Φ-functions 7
1.4.2 Minimal and pruned SSA 8
1.5 SSA construction algorithm motivation 9
1.5.1 Value numbering 10
1.6 Φ-elimination algorithm motivation 10

2 Overview of implemented algorithms **13**

2.1 SSA construction algorithm overview 13
2.2 Φ-elimination algorithm overview 17

3 A new code generation API **21**

3.1 Filling and sealing 23
3.2 Memory 24
3.3 Example code generation 24

4 Implementation **29**

4.1 Implementation of the SSA construction algorithm 29
4.1.1 Appending statements and the algorithm 31
4.1.2 Finalize: From Hack representation to GIMPLE 31
4.1.3 Why use a custom representation? 32
4.2 Implementation of the Φ-elimination algorithm 32

5 Results and discussion **35**

5.1 Applying Hack API to GCC 35

5.1.1 Optimization pass intossa 35
5.1.2 Inserting code 36
5.2 Measuring Φ-elimination effectivity in GCC 37

Conclusion **39**

Bibliography **41**

A Building the modified GCC **43**

Introduction

Before a program can be executed it has to be compiled. Compilation is the process of translating source code written in a programming language to instructions executable by a computer. Compilers are programs that compile other programs.

Compilers are usually able to optimize code during compilation. These optimizations are not done over source code text and only some of them are done over machine code. They are usually done over some intermediate representation [1].

Compilers build data structures over the intermediate representation to help with optimizations. Among others there are the *control flow graph (CFG)* and the *static single-assignment (SSA)* form. Both of these structures are crucial for modern compilers [1].

The *GNU Compiler Collection (GCC)* is currently one of the most popular compilers [2]. It implements a traditional SSA construction algorithm from Cytron, Ferrante, Rosen, Wegman, and Zadeck [3].

Braun, Buchwald, Hack, Leißa, Mallon, and Zwinkau [4] introduced a new algorithm for SSA construction. Compared to the traditional algorithm it does not require the whole CFG to be constructed beforehand. It can therefore be used *while* we are building an intermediate representation and its CFG.

A part of the SSA construction algorithm is a cleanup phase. It deals with Φ-functions which are structures present in SSA form. We expect that this cleanup phase will be useful as a standalone Φ-function optimization algorithm.

The goal of this thesis is to evaluate the utility of the new SSA construction algorithm and the Φ-function optimization algorithm in GCC.

Using the new SSA construction algorithm we design an API for generating intermediate representation in SSA form – more specifically the GIMPLE intermediate language in SSA form. The long-term goal is to eliminate non-SSA GIMPLE from GCC which the API makes possible. That would however be out of scope for a bachelor thesis so instead we implement a basic version of the API and showcase that it can substitute the traditional SSA construction algorithm in simple cases. We also discuss other usecases for the API for the time before the long-term goal is achieved.

As the Φ-function optimization is a part of the SSA construction algorithm

we also implement it. We take this opportunity to measure how effective this algorithm is as a standalone optimization to surprisingly good results.

- Chapter 1 reviews the theory of compilers and the details of GCC needed to understand the thesis.
- Chapter 2 describes the algorithms that we aim to implement.
- Chapter 3 describes the code generation API we designed.
- Chapter 4 describes how we approached the implementation of the API and the algorithms.
- Chapter 5 showcases GCC using the API. Here we also measure the utility of the Φ-function algorithm.

Chapter 1

Background

In this chapter we review the theoretical concepts and GCC codebase details necessary to understand this thesis.

1.1 GIMPLE intermediate language

Compilers use intermediate representation of code for most optimizations. This intermediate representation is called an *intermediate language* (IL). GCC uses three intermediate languages—GENERIC, GIMPLE and RTL. When compiling with GCC we represent the program first in GENERIC, then we convert into GIMPLE, then into RTL and only then we proceed to generating machine code [2]. In this thesis we will be dealing only with the GIMPLE IL.

In GIMPLE, we represent statements by the `gimple` class and its subclasses. There are different types of statements, for example assignment statements, function call statements, return statements, conditional statements and more. See Listing 1

Each statement has at most 3 operands. There are exceptions, for example function calls. We represent an operand of a GIMPLE statement by the `tree`

Listing 1 In order: assignment, call, return and conditional.

```
a = b + 2;
a = fun (a, b, 4);
return a;
if (a < b)
  goto <bb3>;
else
  goto <bb4>;
```

structure. The `tree` structure serves a lot of purposes in GCC. In this thesis we will consider `tree` representation of variables, SSA names (we talk about SSA names later), constants and memory accesses. More details about GIMPLE can be found in *GCC 8.0 GNU Compiler Collection Internals*, Chapter 12 [2].

We call the operands of a statement the *right hand side (rhs)* of the statement. If the statement assigns to a variable or an SSA name x, we call this x the *left hand side (lhs)* of the statement.

1.2 Optimization passes

In GCC the process of compilation is done in steps called *passes* [2, Chapter 9]. A pass gets some representation of the program being compiled as its input and produces a different form of the program. There are optimization passes whose goal is to optimize the program in some way. For example the copy propagation pass, the dead code elimination pass and the full redudancy elimination pass are optimization passes. There are also passes that convert the program from one representation to another—for example from GENERIC to GIMPLE. We say that those passes *generate code*. Some passes may insert fragments of code into the program. That is also code generation.

We define the order in which passes are run in a *pass queue*.

GCC implements both *interprocedural* passes that work on multiple functions at once and *intraprocedural* passes that work on a single function at a time. In this thesis we will only focus on intraprocedural optimizations. When refering to the function that we are currently modifying we will use the name `cfun` since that is the name used in the GCC codebase.

1.3 Control Flow Graph

A control flow graph (CFG) is a standard way to organize itermediate representation.

Definition 1.1 (Basic block [3]). *The statements of a program are organized into (not necessarily maximal)* basic blocks*, where control-flow enters a basic block at its first statement and leaves the basic block at its last statement.*

Definition 1.2 (Control Flow Graph (CFG) [3]). *A* control flow graph *is a directed graph whose nodes are the basic blocks of a program and two additional nodes,* Entry *and* Exit. *There is an edge from Entry to any basic block at which program can be entered, and there is an edge to Exit from any basic block that can exit the program. The other edges of the graph represent transfers of control (jumps) between the basic blocks.*

Listing 2 GIMPLE code

```
M = M % m;
foo = m;
m = M;
M = foo;
```

Listing 3 The same GIMPLE code in SSA form

```
M_5 = M_3 % m_4;
foo_6 = m_4;
m_7 = M_5;
M_8 = foo_6;
```

Definition 1.3 (Successors and predecessors [3]). *For each node X,* successor *of X is any node Y with an edge $X \to Y$ in the graph. Similarly for* predecessors.

1.4 Static Single-Assignment Form

Many optimizations in GCC rely on the program being in static single-assignment form.

Definition 1.4 (Static Single-Assignment Form (SSA) [1]). *An intermediate representation of a program is in* static single-assignment form *if and only if we assign to each variable name only once.*

Since programs usually assign to a variable multiple times we distinguish between different left hand side occurences of a variable by assigning unique *SSA names* to these occurences. Each SSA name corresponds to a variable. However a variable can have multiple SSA names. When talking about SSA names of variables we usually subscript numbers to the names of the variables. See Listings 2 and 3 for examples of code before and after assigning SSA names. Note the two assignments to variable `M`.

1.4.1 Φ-functions

It is possible and quite common that a variable is defined in two different branches of the program. In the example [1] in Listing 4 we have two branches where the variable x gets defined. Each of these definitions of x will get a different SSA name. Which of the SSA names should we use for the x operand in the last statement?

Listing 4 A conditional in non-SSA form

```
if ( flag )
  x = -1;
else
  x = 1;
y = x * a;
```

Listing 5 A conditional in SSA form

```
if ( flag )
  x_1 = -1;
else
  x_2 = 1;
x_3 = PHI(x_1, x_2);
```

In this situation we combine multiple definitions using a Φ-*function*. Operands of Φ-functions are SSA names. A Φ-function evaluates to the operand that corresponds to the path we took to get to the Φ-function. In Listing 5 the Φ-function evaluates to `x_1` if we took the true branch or to `x_2` if we took the false branch.

1.4.2 Minimal and pruned SSA

For a given program it is possible to create different SSA representations. However these representations can differ in the number of Φ-functions. We want the compiler to work as efficiently as possible. Therefore we want to minimize the number of Φ-functions that we work with. Here are the definitions of pruned SSA form and minimal SSA form – restrictions on the number of Φ-functions.

Definition 1.5 (Pruned SSA form [4])**.** *A program is said to be in* pruned SSA form *if each Φ-function (transitively) has at least one non-Φ user.*

Definition 1.6 (Path convergence [4])**.** *Two non-null paths in CFG $X_0, X_1, ..., X_J$ and $Y_0, Y_1, ..., Y_K$ are said to* converge *at a block Z if and only if the following conditions hold:*

$$X_0 \neq Y_0$$

$$X_J = Z = Y_k$$

$$(X_j = Y_k) \Rightarrow (j = J \vee k = K)$$

Definition 1.7 (Necessary Φ-function [4])**.** *A Φ-function for variable v is necessary in block Z if and only if two non-null paths in CFG $X, ..., Z$ and $Y, ..., Z$ converge at a block Z, such that the blocks X and Y contain assignments to v.*

Definition 1.8 (Minimal SSA form [4])**.** *A program with only necessary* Φ*-functions is in* minimal SSA form

Note that the "minimal" SSA form as we have defined it does not have to be truly minimal and that even "necessary" Φ-functions as we have defined them may be redundant in some situations. These are just the standard names for these definitions.

1.5 SSA construction algorithm motivation

We will call the process of converting GIMPLE (or any other IL) into SSA form *SSA construction.* This involves assigning SSA names and computing placements of Φ-functions.

Cytron, Ferrante, Rosen, Wegman, and Zadeck [3] present an efficient SSA construction algorithm. This algorithm is widely known and used [5, 6]. GCC currently implements this algorithm. However, over the years researchers found other SSA construction algorithms. One of them is an algorithm from Braun, Buchwald, Hack, Leißa, Mallon, and Zwinkau [4]. The algorithms from Cytron et al. could be described as operating in forward direction and the algorithm from Braun et al. could be described as operating in backward direction [4] so we will refer to these algorithms as the *forward* algorithm and the *backward* algorithm [4]. These are the only SSA construction algorithms we will consider in this thesis.

Both the forward and the backward algorithms are able to produce SSA form that is minimal and pruned. Both articles contain a proof of this [4, 3]. However, forward algorithm requires the whole input program to already be represented as a CFG in non-SSA form [4]. In contrast, the backward algorithm works even on incomplete CFG. It is therefore possible to use it while still generating code and building CFG.

Our long-term goal is to eliminate usage of non-SSA GIMPLE in GCC. Currently, GCC converts GENERIC into GIMPLE, then it constructs CFG and then it builds SSA. It could be possible to go from GENERIC directly into GIMPLE CFG in SSA form. This would save time during compilation and would make the GCC codebase more elegant. This thesis is an effort towards this goal.

Another application of backward algorithm could be the GCC just-in-time (JIT) compilation framework. JIT compiler usually does not have the whole CFG available. Therefore, using the forward algorithm is not possible and the JIT framework currently does not use SSA. With the backward algorithm it may be possible to use SSA in GCC JIT. That would enable us to use more powerful optimizations in GCC JIT.

We will not refer to the forward algorithm again in the rest of the thesis. Therefore let us switch to calling the backward algorithm (from Braun et al.) just

Listing 6 Local value numbering: Source program.

```
a = 42;
b = a;
c = a + b;
a = c + 23;
```

Listing 7 Local value numbering: SSA form

```
v_1: 42
v_2: v_1 + v_1
v_3: 23
v_4: v_2 + v_3
```

the SSA (construction) algorithm.

1.5.1 Value numbering

The authors of the SSA construction algorithm [4] formulate it not in terms of SSA names but in terms of *value numbers* [7]. Value numbering is a standard technique for eliminating redundancies in a program. We assign numbers to expressions—the same numbers to equivalent expressions [8]. Instead of refering to variables we then refer to value numbers of expressions. The authors of the SSA algorithm use value numbering to model SSA construction. See Listing 6 and Listing 7.

1.6 Φ-elimination algorithm motivation

The authors of the SSA construction algorithm prove that it outputs SSA form that is pruned and minimal [4]. However, this proof assumes that we include a clean up phase. Without the cleanup the algorithm produces minimal SSA on most but not all CFGs[1]. The cleanup phase is handled by another algorithm introduced in the same article. It can be viewed as a part of the SSA construction algorithm or as a separate algorithm. We will call this algorithm the *(strongly-connected component based)* Φ*-elimination algorithm.* As the name suggest it makes use of the concept of *strongly connected components* (SCCs) in graphs.

We implement the Φ-elimination algorithm for multiple reasons. Firstly, we would like our implementation of the SSA algorithm to produce minimal

[1]The algorithm does not create minimal SSA form in the case of *irreducible* CFG. This type of CFG is rare [9] but we would like to handle it nontheless

SSA. Another reason is that we expect that this algorithm will prove useful as a standalone optimization in GCC. When converting to SSA, GCC produces SSA form that is minimal and pruned. However, other optimizations break this property and redundant Φ-functions accumulate over time. The Φ-elimination algorithm could be able to remove a significant portion of them if placed into the right place in the pass queue. In this thesis we measure (see Chapter 5) how many redundant Φ-functions get found by the Φ-elimination algorithm. Aside from being a metric of the effectiveness of this algorithm outside the original context, these measurements may be insightful for understanding how optimization passes in GCC interact.

Chapter 2

Overview of implemented algorithms

In this chapter, we describe the SSA construction algorithm and the Φ-elimination algorithm as they are introduced in Braun, Buchwald, Hack, Leißa, Mallon, and Zwinkau [4].

2.1 SSA construction algorithm overview

See Listings 8 and 9 for the pseudocode of the SSA construction algorithm from the original article [4].

The algorithm is split into multiple functions. The functions `readVariable`, `writeVariable` and `sealBlock` form an API for code generation. The algorithm ensures that the resulting code is in SSA form. It does not actually assign any SSA names. It assigns value numbers instead. However, as we noted in Chapter 1 SSA names and value numbers are analogous.

Be aware that to use the API correctly **we must generate code only by appending statements to basic blocks.** Inserting statements in the middle of a basic block is not allowed. It is a limitation of the algorithm. We did not, however, find it to be a major inconvenience.

For now let us also assume that all the statements we are generating have a left hand side.

The function `readVariable` translates variables to value numbers. Let us assume we want to append a new statement to a basic block. If the statement has any variables as operands, the operands have to be converted to value numbers. Therefore we call `readVariable` on all operands.

Whenever we append a statement we must call the `writeVariable` function to register that this statement is the new definition of its left hand side variable.

Listing 8 SSA construction algorithm pseudocode part 1

```
# local value numbering

writeVariable(variable, block, value):
    currentDef[variable][block] <- value

readVariable(variable, block):
    if currentDef[variable] contains block:
        # local value numbering
        return currentDef[variable][block]
    # global value numbering
    return readVariableRecursive(variable, block)


# global value numbering

readVariableRecursive(variable, block):
    if block not in sealedBlocks:
        # Incomplete CFG
        val <- new Phi(block)
        incompletePhis[block][variable] ← val
    else if |block.preds| = 1:
        # One predecessor: No phi needed
        val <- readVariable(variable, block.preds[0])
    else:
        # Break potential cycles with operandless phi
        val <- new Phi(block)
        writeVariable(variable, block, val)
        val <- addPhiOperands(variable, val)
    writeVariable(variable, block, val)
    return val

addPhiOperands(variable, phi):
    # Determine operands from predecessors
    for pred in phi.block.preds:
        phi.appendOperand(readVariable(variable, pred))
    return tryRemoveTrivialPhi(phi)
```

Listing 9 SSA construction algorith pseudocode part 2

```
# detect and recursively remove a trivial phi function

tryRemoveTrivialPhi(phi):
    same <- None
    for op in phi.operands:
        if op = same || op = phi:
            continue # Unique value or selfreference
        if same != None:
            # The phi merges at least two values: not trivial
            return phi
        same <- op
    if same = None:
        same <- new Undef()
    # Remember all users except the phi itself
    users <- phi.users.remove(phi)
    # Reroute all uses of phi to same and remove phi
    phi.replaceBy(same)

    # Try to recursively remove all phi users,
    # which might have become trivial
    for use in users:
        if use is a Phi:
            tryRemoveTrivialPhi(use)
    return same


# handling incomplete CFG

sealBlock(block):
    for variable in incompletePhis[block]:
        addPhiOperands(variable,
                       incompletePhis[block][variable])
    sealedBlocks.add(block)
```

Keeping track of definitions is crucial to correctly translating variables into value numbers.

How exactly does `readVariable` know which SSA name to choose for which occurence of a variable? Suppose we have a statement s in a basic block b and its operand o. Operand o is an occurence of a variable v. To preserve the original meaning of the program, we want to find the last (in terms of program execution order) assignment a to v (its value number). The algorithm approaches this task by searching backwards from the location of statement s.

Firstly, let us consider what happens in the context of a single basic block. Since a basic block is just a list of statements, we may simply traverse the list backwards until we find a. However, thanks to `writeVariable` we already have a cached. We call the mapping between variables and their latest definition `currDef`. We keep this mapping for each block.

Let us now consider what happens if a is not present in the same basic block as s. In that case we recursively search the predecessors of b. We use the function `readVariableRecursive` for this. Suppose that recursive calls on predecessors all returned an assignment. We only wanted one assignment a but now we have assignments $a_1, a_2, ..., a_n$. Therefore we add a Φ-function $\Phi(a_1, a_2, ..., a_n)$ to b. This Φ-function now serves as a.

But sometimes the Φ-function is not needed—it may be trivial:

Definition 2.1 (Trivial Φ-function [4])**.** *We call a Φ-function p* trivial *if and only if it just references itself and one other value v any number of times.*

Any occurence of a trivial Φ-functions p as an operand of a statement can be replaced by an occurence of v since p always evaluates to v. We therefore remove all trivial Φ-functions as soon as we can. A very common case where we would end up with a trivial Φ-function is when there is only one predecessor. We do not bother to create a Φ in this situation. We just pass on the statement that we got from the recursive call on the single predecessor.

Before we explain how the algorithm handles incomplete CFG, let us introduce the concepts of *filled* and *sealed* blocks. A basic block is filled when no additional statements will be added to it. **We require that a block is filled before any edges leading from it are added**. We seal a block (this is an explicit action) when no additional edges leading into it will be added.

The algorithm is designed to work with incomplete CFG. With incomplete CFG, it is possible that the following situation arises: We are calling `readVariable` on a variable v from a basic block b. Variable v does not have a definition in b and b is not sealed. If b was sealed, we would just query its predecessors and created a Φ-function. However at this moment, we have no way of determining all operands that the Φ-function will have in the final generated code. This situation is quite common. It arises every time we are building a

loop. To solve it we put an operandless Φ-function in b. This Φ-function now defines v for this block. We keep track of operandless Φ-functions of each basic block. When a basic block is sealed and we find that there is an operandless Φ for variable v we fill in its operands. We are able to do that now since all predecessors are now present.

2.2 Φ-elimination algorithm overview

In order for the SSA construction algorithm to produce minimal SSA form on every CFG we have to run the Φ-elimination algorithm. Let us describe how it operates. But first we have to introduce some definitions and a lemma.

Definition 2.2 (Redundant set of Φ-functions [4])**.** *A non-empty set P of Φ-functions is* redundant *if and only if the Φ-functions just reference each other or one other value v.*

Definition 2.3 (Strongly-connected component [10])**.** *Let $\leftrightarrow$ be a binary relation on the vertices of a graph such that $x \leftrightarrow y$ if and only if exists an oriented path both from x to y and from y to x.*

Strongly-connected components (SCCs) *are the subgraphs induced by the equivalence classes of $\leftrightarrow$.*

Definition 2.4 (Data flow graph on Φ-functions)**.** Data flow graph *is a directed graph $G = (V, E)$ where V is the set of all Φ-functions in the program and an edge leads from $u \in V$ to $v \in V$ if and only if u references v.*

Lemma 2.5 ([4])**.** *Let P be a redundant set of Φ-functions with respect to v. Then there is a strongly-connected component $S \subseteq P$ that is also redundant.*

Proof. Consider the condensation P' of P (we contract each SCC into a single vertex). Since P' is acyclic [10] it has a leaf s'. Because s' is a leaf, the Φ-functions in the corresponding SCC S may only refer to v or each other and therefore S is redundant. □

We have now defined which Φ-functions we consider to be redundant. The purpose of the PHI removal algorithm is to remove these Φ-functions. The process makes use of Lemma 2.5.

The general idea is that we search for strongly-connected subgraphs of G which are also redundant and remove them. Once there are no connected redundant subgraphs, by Lemma 2.5 there are no redundant sets of Φ-functions.

For pseudocode of the Φ-elimination algorithm see Listing 10. The algorithm is supposed to be run by calling `removeRedundantPhis` on the set of all Φ-functions in the program.

Listing 10 Φ-elimination algorithm pseudocode

```
proc removeRedundantPhis(phiFunctions):
    sccs <- computePhiSCCs(inducedSubgraph(phiFunctions))
    for scc in topologicalSort(sccs):
        processSCC(scc)

proc processSCC(scc):
    inner <- set()
    outerOps <- set()
    for phi in scc:
        isInner <- True
        for operand in phi.getOperands():
            if operand not in scc:
                outerOps.add(operand)
                isInner <- False
        if isInner:
            inner.add(phi)

    if len(outerOps) = 1:
        replaceSCCByValue(scc, outerOps.pop())
    else if len(outerOps) > 1:
        removeRedundantPhis(inner)
```

The function `removeRedundantPhis` takes a set of Φ-functions as its input. It finds redundant SCCs in the set and removes them from the program. We now describe how the function achieves this. Firstly, the function computes SCCs in the data flow graph induced by the input Φ-functions. Then it processes each SCC in a topological order. This is done by calling `processSCC` on the SCC.

The function `processSCC` takes a set of Φ-functions which form an SCC as its input. Its purpose is to determine if the SCC is redundant (in the context of the whole program). If it is then the function removes the SCC from the program. Otherwise, the function attempts to find and remove redundant SCCs inside the input SCC by calling `removeRedundantPhis`. The algorithm is recursive.

How do we know that an SCC P is redundant? We collect the set of values that do not belong into P but are present as operands of Φ-functions from P. We call this set `outerOps`. If the size of this set is bigger than 1, P is not redundant. If the size of this set is exactly 1, P can be removed. If the set is empty, the corresponding basic blocks are unreachable. The SCC is skipped in this case.

Which Φ-functions from SCC P do we consider when searching for inner SCCs? We consider those Φ-functions that reference only other Φ-functions from P. We collect them into the `inner` set.

What does it actually mean to remove an SCC? Some statements may use

the values computed by the Φ-functions as their operands. If we just deleted the Φ-functions, we would be left with statements with undefined operands. However, note that we only delete redundant sets and that redundant sets of Φ-functions always have a single value v originating outside the set. Therefore, each Φ-function in the set evaluates to v. We can replace all references to the Φ-functions in the program to references to v. This is what the function `replaceSCCByValue` does. Braun et al. [4] do not provide pseudocode for this function. Its meaning is clear.

Chapter 3

A new code generation API

We introduce a new API for generating GIMPLE code in SSA form. We call this API the *Hack API*[1]. We do not expose `readVariable` and `writeVariable` as does the API from the SSA construction algorithm article [4]. Value numbering and other inner workings of the SSA construction algorithm are hidden behind the Hack API. We use the Hack API while building CFG to append statements to basic blocks. The API produces GIMPLE code in SSA form that is minimal and pruned.

It is also possible to modify the API to apply light optimizations to the code while constructing it [4, Section 3.1]. We discuss these on-the-fly optimizations in Chapter 5.

To start generating GIMPLE code with its API, create an instance of `class hack_builder`. Its public methods form the API.

Before we create a statement we first have to represent its operands. We do that using `hvar` structures. There are multiple types of `hvars`:

- `LOCAL` represents a local variable of cfun (the current function we are compiling). It is created by calling the method `new_local`. Specifying a GCC `VAR_DECL tree` means creating a named variable. Otherwise, a GCC type `tree` has to be specified and the result is an anonymous variable.

- `PARAM` is similar to `LOCAL`. It represents a cfun parameter. It cannot be anonymous. We create it by calling `new_param`. We have to specify a GCC `PARM_DECL tree`.

- `INVAR` represents a variable or a constant that we do not assign an SSA

[1] In programmer jargon "hack" usually means a solution that is inelegant or clumsy. We do not want to say that about the API. When deciding how to name the API one of the names of the authors of the SSA construction algorithm simply caught our eye. We did not think of a better name since so we kept this one.

name to. For example we may already have an SSA name and want to use it in the code we are generating. Or— more often—`INVAR` represents a constant like `5`, `0` or `true`. We create it by calling `new_invar`.

- `MEMORY` represents an access to memory. We will discuss memory later in this chapter in Section 3.2.
- `OUTVAR` is used when mixing already existing code and code generated through our API. How this is done will be shown in an example in Section 3.3.

Creating a statement and appending it happens as a single function call. We always specify the basic block to which we are appending. Here are the methods we use to append statements. GIMPLE has many more types of statements. We aimed to implement a subset of GIMPLE that is big enough to showcase the functionality of the API.

```
void append_assign (basic_block bb, enum tree_code code, hvar *left,
                    hvar *op1);
void append_assign (basic_block bb, enum tree_code code, hvar *left,
                    hvar *op1, hvar *op2);
void append_assign (basic_block bb, enum tree_code code, hvar *left,
                    hvar *op1, hvar *op2, hvar *op3);
```

Appends an assignment to variable `left`. The right hand side is an expression with up to three operands. `tree_code` specifies the type of the expression (for more details see *GCC 8.0 GNU Compiler Collection Internals*[2, Section 11.3]).

```
void append_cond (basic_block bb, enum tree_code pred_code,
                  hvar *left, hvar *right);
```

Appends a conditional.

```
void append_return (basic_block bb);
void append_return (basic_block bb, hvar *retval);
```

Append a return statement with or without the return value operand.

```
void append_call_vec (basic_block bb, tree fn, hvar *left,
                      const vec<hvar *> &args);
void append_call_vec (basic_block bb, tree fn,
                      const vec<hvar *> &args);
```

Appends a call to a function. Operands are passed in a vector.

```
hvar *append_outvar (basic_block bb, hvar *local);
```

A virtual statement that does not get translated into GIMPLE. Appending this statement tells the Hack builder to compute an SSA name for an `hvar` even though no GIMPLE statement using the SSA name may be generated. The purpose of these statements will be explained in an example in Section 3.3.

```
hvar *append_handled_component (basic_block bb, tree ref,
                                vec<hvar *> &operands);
```

Another virtual statement that does not get translated into GIMPLE. It will get explained later in Section 3.2.

When we do not have any more statements to append, we call the method `finalize`. Untill this point, `hack_builder` only remembered which statements we wanted to append where. Now it actually creates SSA names and GIMPLE statements and we are left with GIMPLE code in SSA form.

3.1 Filling and sealing

Filling blocks and sealing blocks are concepts which distinguish the Hack API from other code generation APIs. We have to keep these concepts in mind when using the API.

A block is filled when no more statements will get appended to it. We seal a block when no more predecessors will get added. As formulated in the original article [4] sealing is an explicit action. We use the function `set_block_sealed` to seal a block. For Hack API we also decided to make filling an explicit action—a call to `set_block_filled`. We did this to remind programmers to think about filled blocks when using this API.

The original article [4] formulates this rule concerning filled blocks: **We may add successors only to filled blocks.** The motivation behind this rule is that when we seal a block we are sure that its predecessors already contain all their statements and are able to provide variable definitions.

In GCC codebase it is currently much easier to use functions which split blocks and split edges instead of creating blocks and edges manually. Splitting a block means creating two blocks from one and linking them with an edge. Splitting an edge means creating two edges from one and putting a new basic block between them. Both of these operations however potentially create unfiled blocks with successors. That conflicts with the rule. Thankfully we found out that the rule can be made more lenient and allow us to first create the CFG structure and then start appending statements.

We formulate a new version of the rule: **We may only seal blocks whose predecessors are filled**. The same property as with the original rule holds: When we seal a block its predecessors are filled and therefore are able to provide variable definitions.

3.2 Memory

Let us talk about memory accesses. In GIMPLE we represent an access to memory by `ARRAY_REF`, `MEMORY_REF`, `COMPONENT_REF` and other `trees`. The other `trees` are rare so in this thesis we only support the listed ones. For our purposes a memory access has two to three operands[2]. The first operand may or may not be another memory access. Memory accesses can nest into each other this way. Each type of memory access has operands that are and operands that are not important to building SSA.

When we want to append a statement with a memory access in it we first call the function `append_handled_component` on the same block for each memory access. We have to provide the memory access `tree` but with the important operands marked with the `error_mark_node` `tree`. We also have to provide a vector of `hvars` corresponding to the marked operands in order from the `0`th operand to the 2nd from the outermost `tree` to the innermost. If we already have a memory access `tree` with some operands that are not valid SSA values we can use the function `extract_operands_to_be_renamed` that collects the important operands in the correct order and replaces them with `error_mark_nodes`.

3.3 Example code generation

We showcase how to use the Hack API on a function that generates a fragment of code. This function (`gimple_divmod_fixed_value`) already exists in GCC in `./gcc/value-prof.cc`. We rewrote it to use the Hack API.

Suppose that we are compiling a program with a division statement. The divisor is a variable. Further suppose we measured that the divisor has a high probability of being a specific `value`. Suppose we want to divide by a constant whenever we can. Therefore we insert a conditional. If the divisor truly equals `value` we divide by `value`. Otherwise we divide by the divisor variable. We can also do the same with a modulo statement. See Listing 11.

We assume that the CFG is already created and looks as shown in Figure 3.1.

[2]We do not support the last two operands of `ARRAY_REF` that are used for more advanced array accesses.

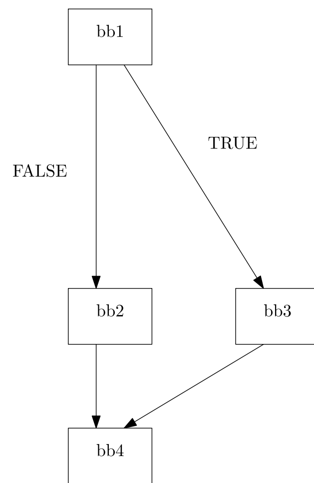

Figure 3.1 CFG structure

Listing 11 Code to be generated

```
if (op2 != value)
  tmp = op1 / op2;
else
  tmp = op1 / value;
```

Listing 12 Initializing the builder and creating hvars

```
hack_ssa_builder builder;
hvar *op1 = builder.new_invar (gimple_assign_rhs1 (stmt));
hvar *op2 = builder.new_invar (gimple_assign_rhs2 (stmt));
hvar *value_as_invar = builder.new_invar (value_as_ssa);
hvar *tmp = builder.new_local (optype);
```

Listing 13 Appending statements

```
/* bb1.  */
builder.set_block_sealed (bb1);
builder.append_cond (bb1, NE_EXPR, op2, value_as_invar);
builder.set_block_filled (bb1);

/* bb2 (false branch).  */
builder.set_block_sealed (bb2);
builder.append_assign (bb2, code, tmp, op1, value_as_invar);
builder.set_block_filled (bb2);

/* bb3 (true branch).  */
builder.set_block_sealed (bb3);
builder.append_assign (bb3, code, tmp, op1, op2);
builder.set_block_filled (bb2);

/* bb4.  */
builder.set_block_sealed (bb4);
hvar *out = builder.append_outvar (bb4, tmp);
builder.set_block_filled (bb4);
```

We initialize the builder and create `hvars`. Since dividend `op1` and divisor `op2` are already SSA values, we create an `INVAR` for each of them. We also create an invar for `value` because it is a constant. We create a `LOCAL` for the quotient `tmp`. Initialization of `hvars` is shown in Listing 12.

Before appending anything to a basic block we mark it as sealed. After we have appended all statements to a basic block we mark it as filled. This way we follow the rule (Section 3.1) that blocks must be sealed only after their predecessors were filled. First we append a conditional statement to basic block `bb1`. Then we append an assignment with division (or modulo) in it to block `bb2` and then block `bb3`. Appending statements is shown in Listing 13.

We want to pass the value of `tmp` to the rest of the program into which we are inserting the code fragment. However the SSA construction algorithm does not know this and will not compute an SSA name for `tmp` at `bb4`. Therefore we append a virtual `OUTVAR` statement to block `bb4` to force the algorithm to compute the SSA name. This is the standard way how we handle inserting code into existing functions using the Hack API.

Now we are done with appending statements. We call the method `finalize`. At this point the builder has placed GIMPLE statements at the locations we specified. We retrieve the SSA that the variable `tmp` has in `bb4` using `ssa_from_outvar`. Now we free the memory taken up by the builder and the Hack representation (this frees even `hvars` so the SSA value has to be retrieved

Listing 14 Finalization

```
builder.finalize ();
tree ret = builder.ssa_from_outvar (out);
builder.release ();
```

at this point). Finalization and memory freeing is shown in Listing 14.

Chapter 4

Implementation

In this chapter we discuss the implementation details of the SSA construction algorithm and the Φ-elimination algorithm.

The SSA construction algorithm is implemented in the

```
./gcc/insert-gimple-ssa.cc
```

file. The Φ-elimination algorithm is implemented in the

```
./gcc/sccp.cc
```

file.

4.1 Implementation of the SSA construction algorithm

The SSA construction algorithm works with value numbers. It would be clumsy to emulate this using the standard structures which represent GIMPLE statements and variables in GCC—the `tree` and `gimple` structures. Instead, before building the final GIMPLE code with the function `finalize`, we use our own representation. We call it the *Hack representation*. This way we avoid the complexity of extending the standard GCC structures and are able to implement the algorithm more elegantly.

Hack representation mimics the GCC's representation of GIMPLE to a degree. Instead of representing variables, constants and memory acceses by different kinds of `trees`, we use `hvars` (*Hack variables*). Instead of representing statements by `gstmts`, we use `hstmts` (*Hack statements*). We call the process of translating Hack statements into GIMPLE statements and translating value numbers into SSA names *commiting the statements* and *commiting the SSA names*.

There are LOCAL, PARAM, INVAR, MEMORY and OUTVAR hvars. For explanation of the types of hvars, see Chapter 3.

Here is a list of the classes representing different types of statements. It bears similarities to the set of GIMPLE statements. In comparison to GIMPLE, Hack representation currently has fewer types of statements. This thesis aims to implement only the generation of a subset of GIMPLE statements that is big enough to showcase the functionality of the SSA algorithm.

- hstmt_with_lhs is an abstract class. Its meaning is explained bellow.
  - hstmt_assign represents an assignment. hstmt_assign mimics the GIMPLE assignment class gassign.
  - hstmt_call represents a call to a function.
  - †hstmt_const is a virtual statement. Statements defining INVARs are not part of the code we are generating. However, when using INVAR as an operand, we need a value number. The value number we use is the value number of this virtual statement. We also use hstmt_consts as default definitions of LOCALs and PARAMs.
  - †hstmt_handled_component is a virtual statement that represents a memory access.
  - hphi represents a Φ-function.
- hstmt_cond represents a condition at the end of basic block.
- hstmt_return represents a return statement.
- †hstmt_outvar is a virtual statement used when mixing already existing code with code newly generated by the API. See Section 3.3 for an example.

Some statements have operands. Operands of statements are always value numbers. **Value numbers are represented as pointers to their defining statements.**

Some statements do not have their GIMPLE counterpart. Those are the *virtual statements* (marked in the hierarchy with †). We do not commit virtual statements to GIMPLE. Each type of virtual statement fulfills a special purpose instead (as noted in the hierarchy).

Only statements that inherit from hstmt_with_lhs may (or may not) have a left hand side. Other statements never have a left hand side. A statement that has a left hand side does not store its value number in any explicit way. Its value number is simply its address in memory. However, the statement stores information about the variable it assigns to in the form of hvar.

4.1.1 Appending statements and the algorithm

How do we keep track of appended statements? We keep a hash map from basic blocks that we encounter to structures of type `hack_bb`. `hack_bb` contains a list of Hack statements and a seperate list of Hack Φ-functions and additional data relating to the specific basic block and the SSA construction algorithm.

When appending a statement we call `read_variable` for each `hvar` operand[1]. The function `read_variable` is able to map `hvars` to value numbers that currently define them. In comparison with the original formulation of `readVariable` we extended the function to also handle `INVAR` and `MEMORY` variables and default definitions of variables. For these special cases an `hvar` sometimes contains a pointer to a special virtual statement. Once we have a Hack statement with value numbers for each of its operands we call `write_variable` which marks down that in this basic block this statement currently defines the relevant `hvar`.

We implemented the algorithm as a recursive algorithm. It would have been better to remove the recursion and use an explicit stack. However, for showcasing that the algorithm can be implemented in GCC this implementation is sufficient.

4.1.2 Finalize: From Hack representation to GIMPLE

Let us describe what happens when the function `finalize` is called.

Firstly, we commit ssa names—we traverse all basic blocks and assign SSA values to statements with a left hand side. We may traverse the blocks in any order[2]. For statements that assign to a `LOCAL` or a `PARAM` we create a new SSA name. For statements that assign to an `INVAR` we do not create a new SSA name since each `INVAR` already has an SSA value (stored in its corresponding `hstmt_const`). `MEMORY` Hack variables and `hstmt_handled_component` are handled analogously. `OUTVAR`s never occur as a left hand side. Once SSA names have been commited, all statements with left hand side have a valid SSA value. Therefore we now have a mapping from value numbers to SSA values.

After SSA names have been commited we commit the statements—we traverse all basic blocks and translate Hack statements into GIMPLE statements. Since we are now able to map value numbers to SSA names we can convert any Hack statement into GIMPLE with SSA values as operands. Therefore we are once again free to travers basic blocks in any order and commit all statements.

---

[1] `read_variable` and `write_variable` correspond to the `readVariable` and `writeVariable` functions from the SSA construction algorithm

[2] This is helpful because we store basic blocks as keys in a hash table so they are not ordered in any meaningful way.

4.1.3 Why use a custom representation?

Before moving on let us consider the choice to create a new representation. When implementing value numbering passes it is a standard practice to use a custom representation. It makes implementing value numbering optimizations easier. In Section 5 we extend the Hack API with an on-the-fly optimization and we expect that more optimizations will get added in future.

We are also able to keep our representation lightweight and allocate it as we see fit. This way on-the-fly optimizations will remove unnecessary statements before they are commited into GIMPLE and save memory and processing time.

4.2 Implementation of the Φ-elimination algorithm

Let us now move on to the implementation of the SCC-based Φ-elimination algorithm. This algorithm serves as a cleanup procedure for the SSA construction algorithm. Without it there are some sidecases where the SSA construction algorithm will not produce the desired (minimal and pruned) SSA form. We also expect the algorithm to prove useful as a standalone optimization.

We implement this algorithm as an optimization pass working with GIMPLE statements. We named this pass *strongly-connected copy propagation (SCCP)*[3].

We could have implemented the algorithm on the Hack representation. The advantage of this approach would be that Φ-functions would get optimized before they got commited into GIMPLE which would save time. The disadvantage is that the Φ-elimination could not be run idependently of the SSA construction algorithm which is something we aim for. We want the Φ-elimination to also serve as a cleanup pass for when other optimization passes break the minimality of SSA form. As we show in Chapter 5 many Φ-functions can be removed this way.

The algorithm requires computing strongly-connected components. We chose to compute SCCs using Tarjan's algorithm [11] because other passes in GCC also implement it and it outputs SCCs in a reverse topological order which we just reverse to get the topological order we need. Tarjan's algorithm is often presented as a recursive algorithm. However, GCC should be able to optimize even massive programs in which case a recursive algorithm could run out of stack memory and we hope that this pass could get integrated into GCC soon. Therefore we opted to remove the recursion using the stack data structure.

[3]We later found that a GCC pass with the same acronym already exists so a different name will have to be chosen before the implementation is integrated into GCC

The original algorithm only considers Φ statements. We decided to broaden its scope and also include copy statements. A copy statements is an assignment which assigns one variable to another. Copy statements can be viewed as Φ-functions with only one operand. This extension does not interfere with the original algorithm and does not introduce any significant slowdown.

`sccp_propagate` is the main function of the implementation. Its input is a GCC vector of all Φ-functions and copy statements in the cfun (the function we are currently compiling). We removed recursion from the algorithm by the same means and for the same reason that we described when talking about the Tarjan's algorithm. The stack that we are using to remove recursion is named `worklist`.

The function `replace_scc_by_value` handles the operation of replacing references to a redundant SCC from other statements. This functions takes advantage of the fact that GCC supports iterating over uses of an SSA name in linear time.

The operation of replacing an SSA name v by another SSA name u is common in GCC. Some GCC data structures need to be updated when that happens so that they keep track of changes correctly. A single function that would execute all the cleanup necessary is currently missing from the GCC codebase. One has to call multiple cleanup functions. This led us to implement functions `cleanup_after_replace` and `cleanup_after_all_replaces_done`. One is supposed to be called after the replacement operation modified a statement. The other is supposed to be called after after a pass is finished with all the replace operations it has to do. These functions are implemented in the `./gcc/tree-ssa-propagate.cc` file.

Chapter 5

Results and discussion

5.1 Applying Hack API to GCC

To showcase our implementation of the Hack code generation API we used it to implement inserting code fragments in the *value profiling* [12] pass and conversion of GIMPLE to SSA GIMPLE.

5.1.1 Optimization pass intossa

The pass *intossa* located in `./gcc/tree-intossa.cc` is responsible for converting non-SSA GIMPLE to SSA form. We modified it so that it optionally uses the Hack API for the conversion. The commandline flag `-fnew-intossa` of the `gcc` binary switches from the original implementation to the Hack API one.

Using the Hack API intossa we successfully ran 11420 out of 23526 GCC testcases from the `execute.exp` suite—about 49%. This suite does not only check that GCC is able to compile the test files but also that the compiled binary runs correctly. These are not bad results considering that we implement only a subset of GIMPLE statements. This shows we have a working prototype of intossa implemented using the Hack API.

To show that the Hack API is capable of doing on-the-fly optimizations we implemented a simple local redundancy elimination. If we are to append an assignment statement s and we have already seen an assignment statement t with the same right hand side in the same basic block we do not append s and instead set t as the definition of the left hand side variable of s. See Listing 15 and Listing 16 for example of how this redundancy elimination works. This example is taken from a short program we were able to compile with the Hack API intossa.

Our implementation of intossa using Hack API is inefficient. Instead of modifying existing GIMPLE statements it removes them and creates new ones.

Listing 15 A fragment of code before redundancy elimination

```
c = a + b;
d = a + b;
e = a + b;
D.2764 = e;
return D.2764;
```

Listing 16 A fragment of code after redundancy elimination

```
c_7 = a_2(D) + b_1(D);
_8 = c_7;
return _8;
```

We expect that with further work it would be possible to modify the Hack API to support in-place modifications.

We also measured how fast the Hack API intossa is compared to intossa using the traditional algorithm. For this measurement we chose the biggest GCC testcase we were able to compile using the Hack API intossa—`PR28071`[1]. Averaged accross multiple runs the new intossa implementation runs for $0.26s$ until `finalize` and then for $0.14s$. The work in `finalize` is mostly allocating GIMPLE statements. The original implementation runs for $0.25s$. Our implementation runs longer. However, if we disregard allocating new GIMPLE statements the new implementation does not introduce any statistically significant slowdown. We can therefore expect that intossa implemented using an extension of the Hack API that would support modifying existing GIMPLE in-place would be comparably fast to the original intossa implementation.

5.1.2 Inserting code

Some passes insert code into the program that is being compiled. We expect that they could profit from being rewritten using our API. We rewrote three functions from the value profiling pass located in `./gcc/value-profiling.cc`:

```
gimple_mod_pow2
gimple_mod_subtract
gimple_divmod_fixed_value
```

generate code using the Hack API. Each of these function inserts GIMPLE code into the program to make it more efficient based on previous profiling. We have

[1] As it still was not big enough we duplicated its code about 30 times.

benchmark	total (1)	removed (1)	% (1)	total (2)	removed (2)	% (2)
deepsjeng	2788	274	9.83	5332	103	1.93
exchange2	4436	766	17.27	9902	316	3.19
gcc	294166	15764	5.36	694294	5171	0.74
leela	6868	666	9.70	5155	37	0.72
mcf	872	120	13.76	1682	25	1.49
omnetpp	50552	3078	6.09	14137	23	0.16
perl	88200	13734	15.57	111122	1219	1.10
x264	28936	2568	8.87	45369	519	1.14
xalancbmk	132536	9524	7.19	236223	3211	1.36
xz	6980	518	7.42	8452	95	1.12

Table 5.1 Removed Φ-functions measurements

already shown one of these modified functions in Section 3.3. We think that after rewriting the functions their implementation is more elegant.

We compiled a program for each of the functions where the modified function successfully inserted the desired code.

If more types of GIMPLE statements are added to the Hack API we expect that more passes that insert code could be rewritten to use the API. For example sanitizers and inlining would profit from the redundancy elimination described in the Section 5.1.1 since those are passes that often insert the same code into the same function multiple times.

5.2 Measuring Φ-elimination effectivity in GCC

To measure how useful is the Φ-elimination algorithm in GCC we ran the standard GCC benchmarks using a modified GCC with the Φ-elimination pass inserted into the pass queue. For each run we measured how many Φ-functions (only Φ-functions, no copy statements) were present before executing the pass and how many of them did the pass identify as redundant. See Table 5.1 for these counts and the percentages of removed statements. We inserted the pass at two different locations in the optimization queue (so over one run of GCC the pass was executed twice). We inserted the pass after early optimizations and after late optimizations.

For the exact placement of the pass in pass queue see the corresponding patch attached to this thesis. Notably the first pass is placed after the `phiopt` and `merge_phi` passes. This means that we really give GCC a chance to remove as many Φ-functions it can before we make the measurements.

The results are surprising. Even though that GCC generates minimal pruned SSA form, after early optimizations are done the SSA form is in a state where we

SCCs	1	2	3	4	5	6	7	8	9	10
50878	50496	299	54	9	15	0	0	3	0	2

Table 5.2 Encountered SCCs measurements

are able to remove a significant portion of Φ-functions—sometimes over 17%.

We expected that the late optimizations should eliminate almost all redundant Φ-functions. However a non-negligible ammount of redundant Φ-functions survived even those optimizations.

With some insight into what the benchmarks compute we may guess that the ammount of redundant Φ-functions correlates with the level of abstraction of the program. For example one of the benchmark is compiling GCC itself. The codebase of GCC was not originally written in C++ but in C—a language with arguably far less abstraction—and GCC's developers aim for efficiency over abstraction. Coincidentaly, GCC has the smallest percentage of redundant Φ-functions found in the first pass. On the other hand the benchmark `deepsjeng` contains a lot of C++ abstraction and the percentage of redundant statements found in the first pass is almost double compared to the GCC benchmark.

We also measured how many redundant Φ-function SCCs of which size we encountered. The resulting counts are summed accross all benchmarks. See the Table 5.2.

From the results it is clear that SCCs containing only a single Φ-function are by far the most prevalent. An SCC of this type is equivalent to a trivial Φ-function from Definition 2.1. Detection of trivial Φ-functions is much easier than detection of redundant SCCs. It is surprising that early optimizations are currently not able to remove these Φ-functions.

The effectiveness of the Φ-elimination algorithm exceeded our expectations. Firstly, this proves the Φ-elimination algorithm worthy of inclusion into GCC. Secondly, this is a motivation to closely investigate early optimization passes in GCC and try to determine why do so many redundant Φ-functions emerge.

Conclusion

The goal of this thesis was to evaluate the utility of the SSA construction algorithm presented by Braun, Buchwald, Hack, Leißa, Mallon, and Zwinkau [4] and its Φ-function cleanup phase as a standalone optimization pass is GCC. To achieve this goal we designed and implemented a basic code generation API using the SSA construction algorithm. We also fully implemented the Φ-function optimization pass.

Our results include:

- We were able to partially rewrite the GCC pass that converts GIMPLE into SSA form to use the new API. We successfully ran a significant amount of standard GCC tests using the modified pass.
- We identified that the API could be useful for inserting code. We showed that this is possible on the value profiling pass.
- We measured the number of Φ-functions that the cleanup pass finds and removes. We found out that GCC optimizations produce a surprising ammount of redundant Φ-functions.

Now that we have designed an API using the new SSA construction algorithms, many opportunities and usecases present themselves. Our long-term goal is to use the API to eliminate non-SSA GIMPLE from GCC. The API has to be extended to handle all types of GIMPLE statements first. The API could also be useful in the GCC JIT framework. We can leverage the fact that the API works with value numbers and implement some on-the-fly optimizations. Inlining and sanitization passes could then use the API when inserting code and would profit from the optimizations.

In conclusion, we showed that both the SSA construction algorithm and its cleanup phase have promising applications in GCC—applications that the article introducing these algorithms does not list. While further work will be needed for the SSA algorithm to reach its full potential, the cleanup algorithm already produces useful results.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Always Learning. Pearson, 2014. ISBN: 9781292024349.

[2] R.M. Stallman and GCC Developers Collective. *GCC 8.0 GNU Compiler Collection Internals*. 12th Media Services, 2018. ISBN: 9781680921878.

[3] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[4] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. "Simple and efficient construction of static single assignment form". In: *Compiler Construction: 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, pp. 102–122.

[5] Diego Novillo. "Design and implementation of Tree SSA". In: *Proceedings of GCC developers summit*. Citeseer. 2004, pp. 119–130.

[6] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE. 2004, pp. 75–86.

[7] Preston Briggs, Keith D Cooper, and L Taylor Simpson. "Value numbering". In: *Software: Practice and Experience* 27.6 (1997), pp. 701–724.

[8] Thomas VanDrunen and Antony L Hosking. "Value-based partial redundancy elimination". In: *International Conference on Compiler Construction*. Springer. 2004, pp. 167–184.

[9] Donald E Knuth. "An empirical study of FORTRAN programs". In: *Software: Practice and experience* 1.2 (1971), pp. 105–133.

[10] M. Mareš and T. Valla. *Průvodce labyrintem algoritmů*. CZ.NIC, z.s.p.o., 2017. ISBN: 9788088168195.

[11] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[12] Z. Dvořák, J. Hubička, P. Nejedlý, and J. Zlomek. “Infrastructure for Profile Driven Optimizations in GCC Compiler”. In: (2002). URL: `http://www.ucw.cz/~hubicka/papers/proj/index.html`.

[13] GCC Developers Collective. *Installing GCC*. 2023. URL: `https://gcc.gnu.org/install/index.html` (visited on 07/20/2023).

Appendix A

Building the modified GCC

To build modified GCC we start with the commit

`d9d6774527bccc5ce0394851aa232f8abdaade4c`

from 2023/04/27. We create the git directory `$GCC_DIR`.

```
mkdir $GCC_DIR
git clone git://gcc.gnu.org/git/gcc.git $GCC_DIR/
cd $GCC_DIR
git checkout d9d6774527bccc5ce0394851aa232f8abdaade4c
```

The SSA construction patches

To add the new code generation API we apply the patch `api.patch`. We can also optionally apply the patch `intossa.patch` to modify the SSA construction pass to support building using the API and the patch `valueprof.patch` to modify value profiling to use the API.

```
git apply api.patch
git apply intossa.patch
git apply valueprof.patch
```

We use the `-fnew-intossa` flag to tell GCC to use the Hack API during SSA construction.

The Φ-elimination patch

To add the new Φ-elimination pass we apply the patch `sccp.patch`.

```
git apply sccp.patch
```

Building GCC

After we have applied all the patches we wish to apply we may proceed to building GCC. First we create a new directory `$BUILD`. The directory `$BUILD` must not be a subdirectory of `$GCC_DIR`.

```
mkdir $BUILD
cd $BUILD
$GCC_DIR/configure
make
```

To compile with the modified GCC we use the following command

```
$BUILD/gcc/xgcc -B $BUILD/gcc/ <options> <source files>
```

It is also possible to install the modified GCC and have it available in `PATH`. The build process and also installation is described in greater detail on the official GCC website [13].