**University of Alberta**

**Library Release Form**

**Name of Author**: Angela Jean Blanch French

**Title of Thesis**: A Study of Later Phase Static Single Assignment in the Open Research Compiler

**Degree**: Master of Science

**Year this Degree Granted**: 2004

Angela Jean Blanch French
P.O. Box 23
Bay Roberts, NL
A0A 1G0

**Date**: ________________

*If you have no voice, scream;*
*if you have no legs, run;*
*if you have no hope, invent.*

- Cirque du Soleil's *Alegria*

**University of Alberta**

# A Study of Later Phase Static Single Assignment in the Open Research Compiler

by

**Angela Jean Blanch French**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2004

**University of Alberta**

**Faculty of Graduate Studies and Research**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Study of Later Phase Static Single Assignment in the Open Research Compiler** submitted by Angela Jean Blanch French in partial fulfillment of the requirements for the degree of **Master of Science**.

José Nelson Amaral (Supervisor)

H. James Hoover

Vincent Gaudet (External)

**Date**: ____________

To the memory of Dr. Cyril P. Coombs,
who valued education as highly as anyone else I have ever known.

# Abstract

Static single assignment (SSA) is an intermediate code representation used in contemporary production compilers. In processor architectures that implement predicated execution, the intermediate code is typically converted out of SSA form before later phases in the compilation process. In such architectures, the elegant framework provided for code optimizations by SSA is not available after predication is used to eliminate conditional expressions. Thus such compilers cannot benefit from SSA in later compiler phases. $\psi$-SSA is a new intermediate representation that allows the maintenance of SSA after if-conversion.

This thesis introduces $\psi$-SSA in a later phase of the Open Research Compiler (ORC). Most traditional SSA algorithms use a worklist to process the nodes in the Control Flow Graph representation of the program when building the SSA form. We propose an improvement to the SSA construction algorithm that reduces both the number of worklists processed as well as the size of the initial set of nodes in some of the remaining worklists. We measure the gains of this improvement in the standard SPEC CINT2000 benchmark suite.

# Acknowledgements

First of all, I would very much like to thank my fiancé. Dave, your constant support in everything I do continues to amaze me. In particular, I want to thank you for your ever-present interest in my work, confidence in my abilities, and encouragement of my dreams. We really are a team.

Next, I want to express my gratitude towards my family. Mom and Daddy, you have always inspired me to strive to do my best. Even though you said I was “on my own” for further education, your consistent interest in what I do is still appreciated. Greg and Debbie, you’ve always been there to help take my mind off school, which has helped so much over the years. Thanks to all of you.

I would also like to mention Arthur Stoutchinin, my external collaborator for this project. Thanks for being so patient with my questions, and helping me to understand your complex piece of work. Allowing me to be a part of your research has been an amazing opportunity.

Finally, my supervisor, Dr. José Nelson Amaral deserves a huge thank you. Nelson, you used the perfect combination of guiding me and giving me freedom to create an excellent working relationship. The immense knowledge you have shared with me has been wonderful. Thanks for always being so patient and supportive.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Static Single Assignment (SSA) is a modern intermediate code representation that aids dataflow analysis by ensuring that each use of a variable can only be associated with one definition. The SSA form is traditionally removed well before the code generator in a production compiler. In particular, the SSA algorithm has not been capable of maintaining its form after predication is used to eliminate conditional branches. Architectures that support predication generally do not use SSA in later phases. Hence, code transformations that occur after predication in supported architectures cannot reap the benefits afforded by the elegant SSA framework.

In this thesis, a method for constructing the SSA form in a later phase of the Open Research Compiler (ORC) is presented. More importantly, a new algorithm for allowing SSA after if-conversion, called $\psi$-SSA, is discussed. $\psi$-SSA combines traditional SSA techniques with ideas incorporating the unique properties of predicated code. Using this method, SSA can be maintained throughout global instruction scheduling. Our initial experiments suggest that building and removing the SSA form in the ORC code generator does not significantly increase compile-time, nor add an overwhelming number of instructions to the baseline results. As well, run time performance is not degraded by the additional work performed during the SSA algorithm.

Current SSA construction algorithms use a worklist method to identify nodes in the Control Flow Graph where $\phi$-functions have to be inserted. This thesis suggests an improvement to this technique whereby individual worklists are compared for equality. If two sets to be iterated over by a worklist are identical, one of the worklists can be eliminated, thus reducing the amount of work performed by the algorithm. Additionally, if one set is a subset of the other, the number of elements in the second worklist can be decreased. On average, 45% of worklist elements to be processed can be saved when the subset relationship is detected.

We begin this thesis with a presentation of background material required to discuss the remainder of the document in Chapter 2. Then, Chapter 3 formally introduces the concept at the heart of this work, SSA. Using the key ideas from Chapter 3, Chapter 4 gives an overview of the current state of the art with respect to SSA. The three main $\phi$-placement techniques are introduced in detail with fully expanded examples. As well, the process of removing the SSA form is discussed, focusing on three techniques currently being studied. The issue of predicated SSA is then the topic of Chapter 5, where the methods discussed are extensions to the traditional SSA algorithms of Chapter 4.

The work of this thesis is performed on the ORC, which is thus the subject of Chapter 6. The ORC's current capabilities are discussed, as well as the modified code generator that results from including $\psi$-SSA. Our suggested improvement to the SSA $\phi$-placement algorithm comes next in Chapter 7, where opportunities for improvement on the ORC are identified. Finally, Chapter 8 presents an experimental evaluation of the modified code generator with respect to compile and run times, instructions inserted, and instructions executed, for a selection of programs from the SPEC CINT2000 benchmark suite. Directions for future work are discussed in Chapter 9, with conclusions tying the entire thesis together found in Chapter 10.

## 1.1 Contributions of this Thesis

This thesis is a comprehensive study of the effects of SSA in a later phase of the compilation process. The major contributions of this thesis include:

1. A detailed examination of current SSA algorithms, using step-by-step examples to illustrate their behaviour. SSA methods for predicated code are also discussed. Particular attention is given to the $\psi$-SSA technique, which is examined with examples.

2. A proof showing that the iterated join set of a set of nodes can be calculated as the union of the iterated join sets of its partitions.

3. An improvement to the standard $\phi$-placement algorithm, based on the proof from item 2, that reduces both the number of worklists processed as well as the

size of the initial set of nodes in some of the remaining worklists. We quantify this enhancement using a selection of the SPEC CINT2000 benchmark suite.

4. An implementation of SSA in a later phase of the ORC. The modified ORC code generator includes an implementation of $\psi$-SSA, which extends the SSA form through global instruction scheduling.

5. An experimental evaluation with original measurements for the later phase SSA's performance. We focus on the effects of compile and run times, the number of instructions inserted, and the number of those executed using the later phase SSA algorithm.

# Chapter 2

# Background Material

Most definitions found in this section are paraphrased from Aho *et al.* [1]. Exceptions are noted.

**Definition 1.** A *basic block* is formed by a sequence of consecutive statements in which flow of control enters at the beginning and only leaves at the end.

**Definition 2.** A *control flow graph, CFG* $= G(V, E)$, is a directed graph representing the flow of control in a program, where $V$ is a set of basic blocks and an edge $(B_i \rightarrow B_j) \in E$ indicates that the execution of the program may be transferred from the basic block $B_i$ to the basic block $B_j$.

In Figure 2.1, $B_0$ is the start node and $B_6$ is the end node of the CFG. Nodes $B_2$ and $B_3$ are *successors* of node $B_1$ and *predecessors* of node $B_4$.

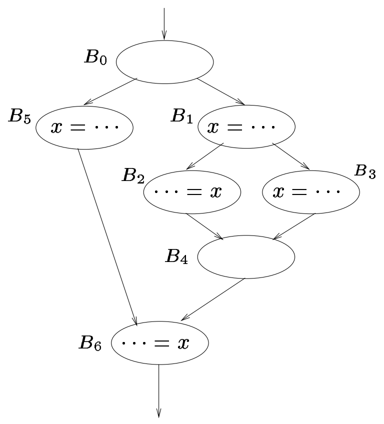


Figure 2.1: A control flow graph

**Definition 3.** Within a basic block, there is a *point* between any two consecutive statements, as well as a point before the first statement, and a point after the last statement.

Figure 2.2 shows some points for the CFG of Figure 2.1. Note that the edge from node $B_1$ to node $B_3$ contains 2 points.

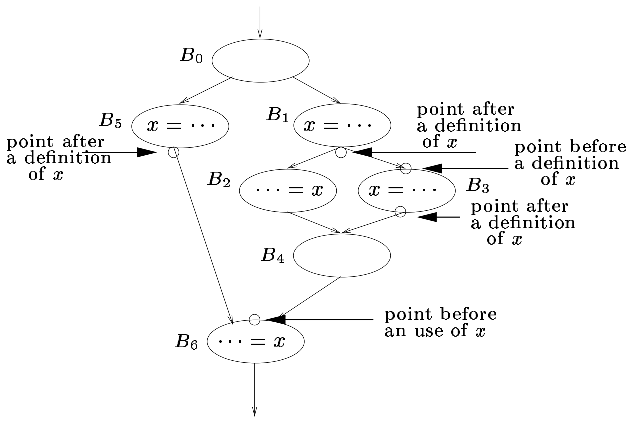

$B_0$
$B_5$
$x = \cdots$
point after a definition of $x$
$B_1$
$x = \cdots$
point after a definition of $x$
point before a definition of $x$
$B_2$
$\cdots = x$
$x = \cdots$
$B_3$
point after a definition of $x$
$B_4$
point before an use of $x$
$B_6$
$\cdots = x$

Figure 2.2: Points in a CFG

**Definition 4.** A *path* from $p_1$ to $p_n$ is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each $i : 1 \leq i \leq n-1$, either

*1.* $p_i$ is the point immediately preceding a statement and $p_{i+1}$ is the point immediately following that statement in the same block; *or*

*2.* $p_i$ is the end of some block and $p_{i+1}$ is the beginning of a successor block.

We often refer to paths in terms of the basic blocks in which their points appear. Given a collection of points such that $p_i \in B_i$, $p_j \in B_j$, ..., $p_n \in B_n$, then a path that includes $p_i, p_j, \ldots, p_n$ may be referred to as the path $B_i, B_j, \ldots, B_n$.

**Definition 5.** A *definition* of a variable $x$ is a statement that assigns a value to $x$.

In the CFG in Figure 2.1, the statements in basic blocks $B_1$, $B_3$, and $B_5$ are all definitions of $x$.

**Definition 6.** Let statement $S_i$ define a variable $x$ and statement $S_j$ have $x$ as an operand. If there is a path $P$ from $S_i$ to $S_j$ that contains no other definitions of $x$, then $S_j$ *uses* the value of $x$ defined by $S_i$.

The statement in node $B_2$ of the CFG in Figure 2.1 uses the value of $x$ defined in node $B_1$, since the path from $B_1$ to $B_2$ contains no other definitions of $x$. We say that the definition of $x$ from $B_1$ *reaches* $B_2$. If another definition did occur along the single path from $B_1$ to $B_2$, the definition found in $B_1$ would be *killed.*

**Definition 7.** A statement $S_i$ that defines a variable $x$ *kills* all previous definitions of $x$ that reach $S_i$ along the paths that include $S_i$.

In Figure 2.1, the definition of $x$ in node $B_1$ is killed along the path $B_1, B_3, B_4$ since $B_3$ contains a definition of $x$, but is not killed along the path $B_1, B_2, B_4$.

When multiple definitions of a variable $x$ appear in a CFG, we are interested in the points that a given definition of $x$ reaches.

**Definition 8.** Suppose that a variable $x$ is defined in a statement $S_i$ in a node of a $CFG = G(V, E)$. We say that $S_i$'s definition of $x$ is *live* at point $p_j$ in $G$ if:

*1.* there is at least one path from $S_i$ to $p_j$ in which $x$ is not killed; *and*

*2.* there exists a path from $p_j$ to an use of $x$ that contains no definition of $x$ [33].

**Definition 9.** Two variables $x$ and $y$ in a program *interfere* if there exists a point $p_i$ in the graph in which both variables are live.

**Definition 10.** The *live-in* set of a basic block $B_i$ (denoted $L_{in}(B_i)$) is the set of all variables live on entry to $B_i$, while the *live-out* set of $B_i$ (denoted $L_{out}(B_i)$) is the set of variables live upon exit from $B_i$.

We want to know how individual CFG nodes relate to each other. The concept of *dominance* is important because it enables the compiler to prove that some definitions cannot reach some uses of a variable.

**Definition 11.** Suppose $B_0$ is the start node of a $CFG = G(V, E)$. Consider two nodes $B_i$ and $B_j$. If every path in $G$ from $B_0$ to $B_j$ goes through $B_i$, then $B_i$ *dominates* $B_j$. Every node dominates itself. If $B_i$ dominates $B_j$ and $B_i \neq B_j$, then $B_i$ *strictly dominates* $B_j$ [18].

In the CFG of Figure 2.1, node $B_0$ dominates every node in the graph. Node $B_1$ dominates nodes $B_1$, $B_2$, $B_3$, and $B_4$. Node $B_1$ does not dominate node $B_6$ because there is another path from $B_0$ to $B_6$, namely the path $B_0, B_5, B_6$. A convenient representation of the dominance relationships among nodes in a CFG is given by the *dominator tree.*

If there exists a downward path $P$ from node $B_i$ to $B_j$ in a directed graph, then $B_i$ is an *ancestor* of $B_j$ and $B_j$ is a *descendant* of $B_i$. These relationships are *strict* if $B_i \neq B_j$.

**Definition 12.** Suppose $B_0$ is the start node of a $CFG = G(V, E)$. The corresponding *dominator tree* for $G$ has $B_0$ as its root, and each node $B_i$ dominates its descendants in the tree.

The dominator tree corresponding to the CFG in Figure 2.1 is given in Figure 2.3. Nodes $B_2$, $B_3$ and $B_4$ are *children* of node $B_1$ and $B_0$ is the *parent* of nodes $B_1$, $B_5$ and $B_6$. The terms parent and child will be reserved for the dominator tree, while predecessor and successor will reference the CFG. Also, the terms descendant and ancestor will be used in conjunction with the dominator tree. In Figure 2.3, node $B_0$ is an ancestor of every other node, while node $B_2$ is a strict descendant of nodes $B_0$ and $B_1$.

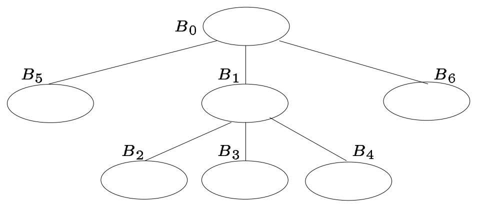


Figure 2.3: A dominator tree

We can also associate a *level* with nodes in the dominator tree.

**Definition 13.** A *level number* for a node $B_i$ in the dominator tree (written $B_i.level$) is the depth of $B_i$ from the root of the tree [32].

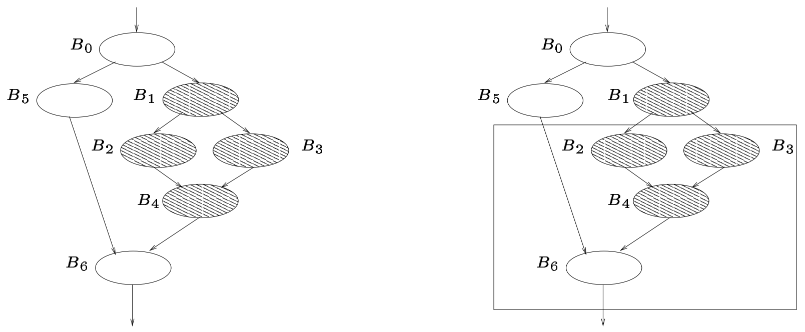
In Figure 2.3, $B_0.level = 0$, $B_1.level = B_5.level = B_6.level = 1$, and $B_2.level = B_3.level = B_4.level = 2$.

**Definition 14.** Suppose that $B_0$ is the start node of a $CFG = G(V, E)$. Consider a node $B_i$ in $G$. The *immediate dominator (idom)* of $B_i$ is the last strict dominator of $B_i$ on any path from $B_0$ to $B_i$ [18].

The children of a node $B_i$ in a dominator tree are all immediately dominated by $B_i$. In Figure 2.3, node $B_0$ immediately dominates nodes $B_1$, $B_5$ and $B_6$. Node $B_1$ immediately dominates nodes $B_2$, $B_3$, and $B_4$.

**Definition 15.** Consider a node $B_i$ in a $CFG = G(V, E)$. The *dominance frontier (DF)* of $B_i$ is the set of all nodes $B_j \in V$ in $G$ such that $B_i$ dominates a predecessor of $B_j$, but $B_i$ does not strictly dominate $B_j$ itself [18].

There are different ways to find the dominance frontier of a CFG node $B_i$. We can start by identifying all the nodes that $B_i$ dominates. These nodes are found by searching the subtree rooted at $B_i$ in the dominator tree. For example, to find the dominance frontier of $B_1$ in the CFG of Figure 2.1, we start by finding the nodes that are dominated by $B_1$, as shown in Figure 2.4(a). We now want to check the successors of these nodes. The set of successors is indicated by the rectangle in Figure 2.4(b). We are looking for successors that are not themselves strictly dominated by $B_1$. Thus, $DF(B_1) = \{B_6\}$.



(a) Nodes dominated by $B_1$ (b) Successors of nodes dominated by $B_1$

Figure 2.4: Finding a dominance frontier

Another computation method for finding the dominance frontier requires the *local dominance frontier* and the dominance frontier *passed* to a node $B_i$ from nodes that $B_i$ immediately dominates.

**Definition 16.** Consider a node $B_i$ in a $CFG = G(V, E)$. The *local dominance frontier ($DF_{local}$)* of $B_i$ is the set of successors of $B_i$ that are not strictly dominated by $B_i$ [18].

Consider node $B_2$ from Figure 2.1. We know from the dominator tree in Figure 2.3 that $B_2$ does not strictly dominate any nodes. The set of successors of $B_2$ is shown by the rectangle in Figure 2.5. The local dominance frontier can then be found by subtracting the set of nodes strictly dominated by $B_2$ from the set of successors of $B_2$. Since $B_2$ does not strictly dominate any node, the local dominance frontier of $B_2$ is just its set of successors. Thus, $DF_{local}(B_2) = \{B_4\}$. Similarly, $DF_{local}(B_3) = \{B_4\}$.

**Definition 17.** Consider a node $B_i$ in a $CFG = G(V, E)$. The dominance frontier of $B_i$ that can be *passed up ($DF_{up}$)* to the immediate dominator of $B_i$ is the set

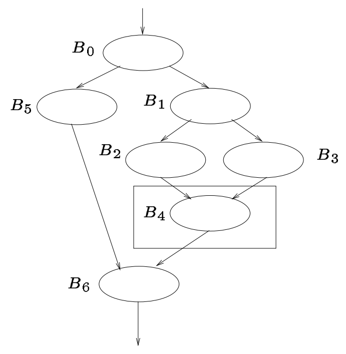
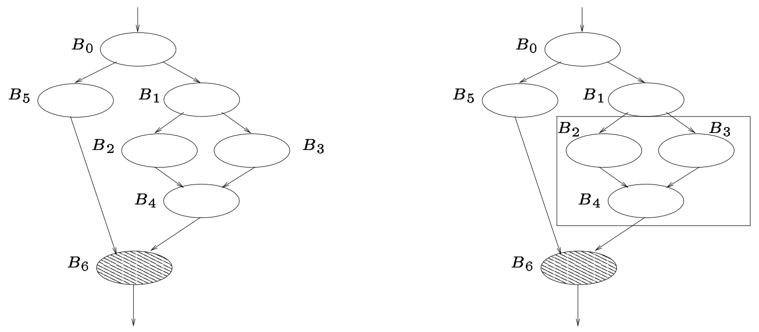


Figure 2.5: Finding a local dominance frontier

of nodes in the dominance frontier of $B_i$ that are not strictly dominated by the immediate dominator of $B_i$ [18].

Consider node $B_4$ of Figure 2.1. We want to compute $DF_{up}(B_4)$. From Figure 2.3, the immediate dominator of node $B_4$ is node $B_1$. We need $DF(B_4)$. From Figure 2.3, we know that node $B_4$ only dominates itself. The only successor of $B_4$ is $B_6$. Since $B_6$ is not strictly dominated by $B_4$, $DF(B_4) = \{B_6\}$, shown in Figure 2.6(a). We also need the set of nodes strictly dominated by $idom(B_4) = B_1$, as indicated by the rectangle in Figure 2.6(b). We are looking for shaded nodes not found in this rectangle (*i.e.*, nodes in $DF(B_4)$ that are not strictly dominated by $B_1$). Thus, $DF_{up}(B_4) = \{B_6\}$.



(a) Dominance frontier of $B_4$ (b) Nodes strictly dominated by $B_1$

Figure 2.6: Finding a dominance frontier passed up to an immediate dominator

The dominance frontier can also be computed using Equation 2.1 [18].

$$DF(B_i) = DF_{local}(B_i) \cup \bigcup_{B_j \in Children(B_i)} DF_{up}(B_j) \tag{2.1}$$

Recall that the children of a node are found by looking at the dominator tree.

Using Equation 2.1, the dominance frontier of node $B_1$ in Figure 2.1 can be calculated:

$$\begin{aligned} DF(B_1) &= DF_{local}(B_1) \cup \bigcup_{B_j \in Children(B_1)} DF_{up}(B_j) \\ &= DF_{local}(B_1) \cup (DF_{up}(B_2) \cup DF_{up}(B_3) \cup DF_{up}(B_4)) \end{aligned}$$

The local dominance frontier of $B_1$ is the set of nodes strictly dominated by $B_1$ subtracted from the set of successors of $B_1$. Thus, $DF_{local}(B_1) = \{B_2, B_3\} - \{B_2, B_3, B_4\} = \emptyset$.

For the dominance frontiers being passed up to $B_1$, it has already been shown that $DF_{up}(B_4) = \{B_6\}$. Nodes $B_2$ and $B_3$ do not pass up anything to $B_1$. Since each of $B_2$ and $B_3$ only dominates itself, and their common successor is node $B_4$, $B_4$ is the only element in each of their dominance frontiers. $B_4$ itself is strictly dominated by node $B_1$, and it therefore does not contribute to the dominance frontier of $B_1$. Thus, $DF_{up}(B_2) = DF_{up}(B_3) = \emptyset$.

The final equation then becomes

$$\begin{aligned} DF(B_1) &= DF_{local}(B_1) \cup (DF_{up}(B_2) \cup DF_{up}(B_3) \cup DF_{up}(B_4)) \\ &= \emptyset \cup (\emptyset \cup \emptyset \cup \{B_6\}) \\ &= \{B_6\} \end{aligned}$$

The result of Equation 2.1 is the same as was computed using Definition 15.

# Chapter 3

# Static Single Assignment

Within compiler research, much work has been done towards effective code analysis and optimization techniques. Traditionally achieved with def-use and use-def chains, code analysis techniques have matured. Now, methods for understanding and improving code focus on the relationships among individual statements and basic blocks [25]. Briggs *et al.* argue that the *static single assignment* (SSA) form is a sparse representation of these relationships [7].

**Definition 18.** A program is in *static single assignment* form if each variable is defined only once.

An SSA form is attractive for compiler code analysis because it reduces the complexity of dataflow analysis. In SSA each variable has a single definition, thus whenever an use of the variable is encountered, there is only one place in the code where the value consumed by that use could have been produced. Allen and Kennedy claim that the most important benefit of the SSA form is the improved performance of optimization algorithms such as constant propagation, forward substitution of expressions, and induction-variable substitution [3]. In particular, an entire category of dependences that arise from reusing variables (resulting in an anti-dependence) or reassigning variables (resulting in an output dependence) can be eliminated, called *false dependences*. Then, the program analysis is left only with true dependences (arising from flow dependences that cannot be eliminated by SSA) with which to contend [39].

Consider the sample code in Figure 3.1. The code shown in Figure 3.1(a) is not in SSA form, since there are two definitions of $x$ and $y$. In this example, a simple renaming of variables produces the SSA form shown in Figure 3.1(b).

Now consider the code of Figure 3.2. The code in Figure 3.2(a) is not in SSA

```
$x \leftarrow 3;$
$y \leftarrow x;$
$x \leftarrow 4;$
$y \leftarrow x;$
```

(a) Non-SSA form

```
$x_1 \leftarrow 3;$
$y_1 \leftarrow x_1;$
$x_2 \leftarrow 4;$
$y_2 \leftarrow x_2;$
```

(b) SSA form

Figure 3.1: Simple conversion into SSA

form, since two definitions of $x$ exist in two distinct control flow paths. By renaming the variables, the code of Figure 3.2(b) is produced.

```
if $x > a$
        $x \leftarrow a;$
else
        $x \leftarrow b;$
$y = x;$
```

(a) Non-SSA form

```
if $x_1 > a$
        $x_2 \leftarrow a;$
else
        $x_3 \leftarrow b;$
$y_1 = ?;$
```

(b) Partial SSA form

```
if $x_1 > a$
        $x_2 \leftarrow a;$
else
        $x_3 \leftarrow b;$
$y_1 = \phi(x_2, x_3);$
```

(c) SSA form

Figure 3.2: Conversion into SSA form

In the final statement of Figure 3.2(b), the value of $x$ assigned to $y_1$ depends on which path is executed at runtime. The potential for an use to be associated with more than one definition occurs at the first node that belongs to two distinct paths in the CFG. Such nodes are called *join nodes* [19].

Alpern and Rosen first introduced a *$\phi$-function*, which is an abstraction used in the join node, to "decide" which definition to use [4, 30]. Figure 3.2(c) shows the example code in SSA form.

$\phi$-functions are found in the SSA intermediate code representation, and are not executable. Inserting $\phi$-functions can be done trivially by determining every variable used in a join node. Let $x$ be a variable used in a join node $B_j$. Then we need to look at definitions of $x$ that are live on entry to $B_j$. A $\phi$-function can be inserted at the point following each such definition of $x$. However, the number of $\phi$-functions that are actually required can be much smaller than those inserted by this method. Inserting unnecessary $\phi$-functions increases the compilation time.

Let $x$ be a variable defined in two or more basic blocks in a $CFG = G(V, E)$. $S_\phi(x)$ is defined as the minimum set of join nodes in $V$ that must receive a $\phi$-function for $x$. A method for computing $S_\phi(x)$ is required. Let $A(x)$ be the set of nodes in $V$

that contain a definition of $x$. Clearly, $DF(A(x)) \subset S_\phi(x)$. However, a $\phi$-function is itself a definition of $x$, therefore an iterated method to compute $S_\phi(x)$ is needed. Based on this formulation, the notion of dominance frontiers from Chapter 2 can be extended to sets of nodes.

When constructing the SSA form of a program, if a variable $x$ has multiple definitions, it is desirable to insert a collection of $\phi$-functions for $x$ at a time instead of just a single $\phi$-function. We therefore want to analyze sets of nodes [18]. Let $X$ be a set of CFG nodes. Then,

$$DF(X) = \bigcup_{B_i \in X} DF(B_i) \tag{3.1}$$

In Section 4.1.1, we will examine the relationship between the *iterated dominance frontier* and where $\phi$-functions should be placed.

**Definition 19.** Given a set of CFG nodes $X$, the *iterated dominance frontier ($DF^+$)* of $X$ is the limit of the following sequence [18]*:*

$$\begin{aligned} DF_1 &= DF(X) && (3.2) \\ DF_{i+1} &= DF(X \cup DF_i) && (3.3) \end{aligned}$$

We make a key assumption during the analysis of a CFG. To ensure that we never have a variable that is used without being previously defined, we assume that all variables are defined in the start node of the CFG. Code analysis is therefore simplified, as we can always assume there is a single definition with which to associate an use.

**Definition 20.** Suppose $X$ is a subset of nodes in a $CFG = G(V, E)$ such that the start node is in $X$. The *join set ($J$)* of $X$ is then the set of all nodes $B_j \in V$ for which distinct nodes $B_i, B_k \in X$ exist where a pair of paths $B_i, \ldots, B_j$ and $B_k, \ldots, B_j$ intersect only at $B_j$ [19].

**Definition 21.** Given a set of CFG nodes $X$, the *iterated join ($J^+$)* of $X$ is the limit of the increasing sequence [18]*:*

$$J_1 = J(X) \tag{3.4}$$

$$J_{i+1} = J(X \cup J_i) \tag{3.5}$$

Cytron *et al.* showed that $S_\phi(x) = J^+(A(x))$, *i.e.*, the minimum set of nodes that require a $\phi$-function for a variable $x$ is the iterated join of the set of nodes that define $x$ [19].

Many algorithms have been developed for constructing the SSA form of a program. Among these algorithms, techniques for $\phi$-function placement and variable renaming have been developed. However, since the $\phi$-function has yet to be found in an instruction set of a machine architecture, the "function" is still not executable. In fact, an instruction that decides which control path was taken is unlikely to exist. Because the function cannot be executed, it must be eliminated before code generation, since there is no code corresponding to the $\phi$-function. Therefore, $\phi$-function removal methods are also of interest. The algorithms for all the phases of SSA construction and removal are presented in Chapter 4.

# Chapter 4

# SSA Algorithms

## 4.1 Algorithms for $\phi$-Function Insertion

The insertion of $\phi$-functions is widely thought to be the core of the SSA construction problem. Several algorithms were proposed for deciding where $\phi$-functions should be placed.

### 4.1.1 Cytron *et al.*

The primary $\phi$-placement method still used in many compilers was presented by Cytron *et al.* in 1989 [18], and further elaborated on in 1991 [19]. This method uses dominance frontiers to determine where the $\phi$-functions should be placed. The relationship between dominance frontiers and $\phi$-functions is established by Theorem 1 [19].

**Theorem 1.** *The set of nodes that need $\phi$-functions for any variable $x$ is the iterated dominance frontier $DF^+(A(x))$. Equivalently,*

$$J^+(A(x)) = DF^+(A(x)) \tag{4.1}$$

Cytron *et al.* compute the dominance frontiers needed for their $\phi$-placement algorithm using Equation 2.1.

The algorithm takes a $CFG = G(V, E)$ and $V$ as input. Also required for each node $B_i \in V$ is $DF(B_i)$, as well as $A(x)$ for each variable $x$ in $G$. The main loop of this worklist algorithm iterates for every variable. For each variable, a worklist, $W$, represents the nodes being processed. Suppose an iteration of the algorithm's main loop is working on a variable $v$. $W$ is initialized to $A(v)$. Then, if $B_i \in W$, a $\phi$-function is inserted in every $B_j \in DF(B_i)$, and $B_i$ is removed from $W$. Recall that a node with a $\phi$-function for $v$ is itself in $A(v)$. Thus for every $B_i \in W$, each

```
$y$ live-in
if ($y > 1$)
    $x = 2$
else if ($y < 1$)
    $x = 3$
    else $x = 10$
for ($i = 0; i < y; i++$){
    if ($x > y$)
      $x++$
    else
      $x--$
}
return($x + y$)
$x, y$ are live-out
```

Figure 4.1: Pseudo-code for the running example

$B_j \in DF(B_i)$ is also relevant, and therefore included in $W$. The algorithm ends when $W = \emptyset$.

Consider Figure 4.2, the CFG for the running example in Figure 4.1. The only non-empty dominance frontiers for this example are given in Figure 4.3. Let $S_\phi(x)$ be the set of nodes that are assigned $\phi$-functions for variable $x$.

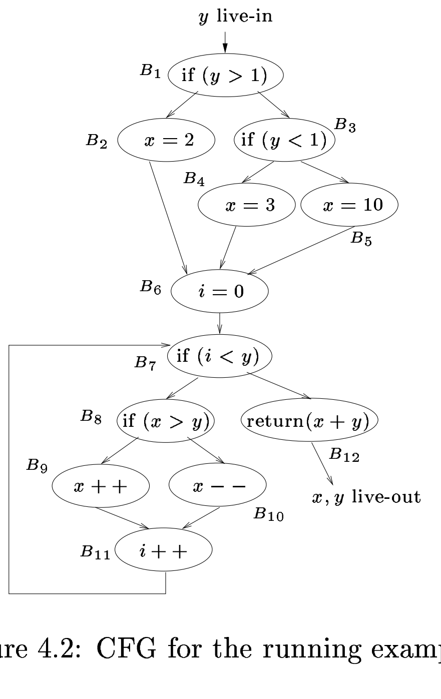


Figure 4.2: CFG for the running example

The loop iteration for $x$ begins by initializing $W = A(x) = \{B_2, B_4, B_5, B_9, B_{10}\}$. Based on the dominance frontiers of these nodes, $S_\phi(x) = \{B_6, B_{11}\}$, and nodes $B_2$,

| | |
|---|---|
| $DF(B_2) = \{B_6\}$ | $DF(B_9) = \{B_{11}\}$ |
| $DF(B_4) = \{B_6\}$ | $DF(B_{10}) = \{B_{11}\}$ |
| $DF(B_5) = \{B_6\}$ | $DF(B_{11}) = \{B_7\}$ |

Figure 4.3: Dominance frontiers for the running example

$B_4$, $B_5$, $B_9$ and $B_{10}$ are removed from $W$. Then each of nodes $B_6$ and $B_{11}$ need to be analyzed, and are thus added to $W$. Since $DF(B_{11}) = \{B_7\}$, node $B_7$ is added to $S_\phi(x)$ and nodes $B_6$ and $B_{11}$ are eliminated from $W$. Node $B_7$ must itself be processed, and is appended to $W$. Since $DF(B_7) = \emptyset$, $B_7$ is taken from $W$ and there are no additional nodes to include in $W$. Thus, $S_\phi(x) = \{B_6, B_7, B_{11}\}$, and $\phi$-functions for $x$ can be added to nodes $B_6$, $B_7$ and $B_{11}$. With $W = \emptyset$, this loop iteration is complete. The main loop will then proceed for every variable in the CFG.

Cytron and Ferrante continued their work in 1995 with improvements to their original algorithms [21]. In particular, the new work avoids computing all the dominance frontiers by producing an order to determine the entire $DF$ relation. Using the order ensures that no elements of the $DF$ relation will be skipped. They focus on two cases. The more general case checks, for an edge $(B_i \rightarrow B_j)$ in the CFG, nodes in the dominator tree between the immediate dominator of $B_j$ and $B_i$, which have been established in the $DF$ relation. The order used is a reverse depth-first numbering, hence nodes are processed if their immediate dominators have decreasing depth-first numbers.

The alternate case encompasses nodes that are siblings in the dominator tree. The pre-determined order from the general case is not applicable, since both nodes have the same immediate dominator. A new relationship is needed [21].

**Definition 22.** Consider a node $B_i$. The *equidominates* of $B_i$ are those nodes with the same immediate dominator as $B_i$. Equivalently,

$$equidom(B_i) = \{B_j \mid idom(B_j) = idom(B_i)\} \tag{4.2}$$

The equidominates are partitioned into blocks of nodes that are contained in each other's iterated dominance frontiers. However the cost of computing individual dominance frontiers is avoided. The required order is then computed based on the edges between equidominates.

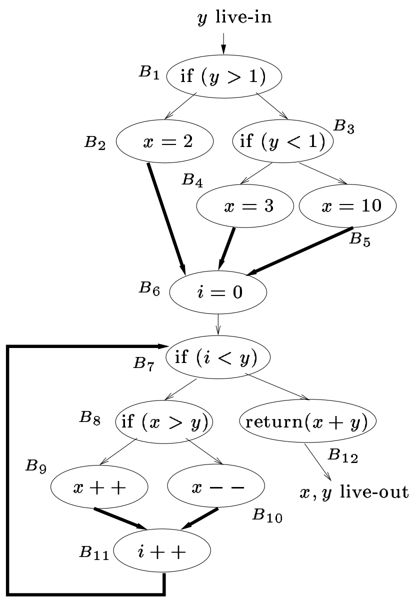


Figure 4.4: J-edges of the CFG for the running example

This method avoids the computation of all dominance frontiers, and also the recursive iteration through the nodes in the dominance frontiers of the worklist nodes.

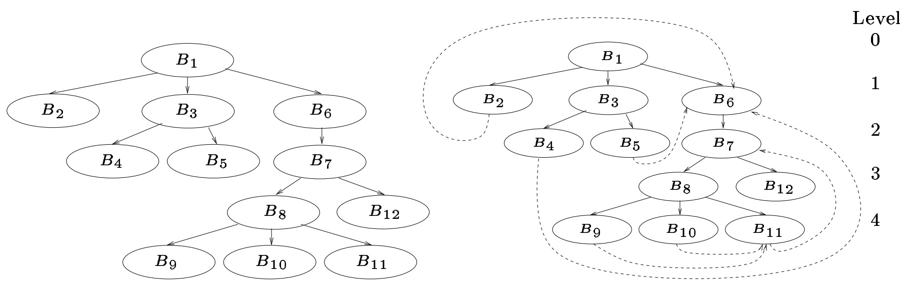
The final form that Cytron and Ferrante have presented is *pruned* SSA [12]. This method only places a $\phi$-function for $x$ at a join node if $x$ is used within or after the join node, *i.e.*, $x$ is live at the entry point of the join node. This strategy differs from the original algorithm, which places $\phi$-functions at all join nodes.

### 4.1.2 Sreedhar and Gao

The next $\phi$-placement method was introduced in 1995 by Sreedhar and Gao [32]. This algorithm requires the construction of a *DJ-graph*, a modification of the traditional dominator tree. The DJ-graph contains all dominator tree edges (referred to as *D-edges*), as well as a set of *J-edges* [32].

**Definition 23.** An edge $(B_i \to B_j)$ in a $CFG = G(V, E)$ is a *join edge (J-edge)* if $B_i$ does not strictly dominate $B_j$.

A DJ-graph can be constructed by inserting join edges from the CFG into the dominator tree. Figure 4.4 indicates the J-edges of the CFG from Figure 4.2 in bold print. Figure 4.5(b) shows the DJ-graph corresponding to the dominator tree of the CFG shown in Figure 4.2. J-edges are indicated by dotted lines in the graph. Also given in the figure are the node levels.



(a) Dominator tree (b) DJ-graph

Figure 4.5: Constructing the DJ-graph for the running example

This algorithm takes as input a DJ-graph and a subset of CFG nodes, $N_\alpha$, and returns $DF^+(N_\alpha)$. It begins by initializing an array $PB$ to $N_\alpha$, with indices based on the level numbers of the individual nodes.[1] The start level is set to be the highest level. Then, each level represented in the $PB$ is processed by visiting each node in that level stored in $PB$. Say node $B_i$ of level $l$ is the current node being processed. For each successor $B_j$ of $B_i$, if $(B_i \rightarrow B_j)$ is a J-edge, then $B_j$ is included in $DF^+$ and is placed in $PB$. If $(B_i \rightarrow B_j)$ is a D-edge, $B_j$ is recursively processed.

Consider the example in Figure 4.2. Let $N_\alpha = A(x) = \{B_2, B_4, B_5, B_9, B_{10}\}$. We then initialize $PB = N_\alpha$. Processing higher level nodes first means nodes $B_9$ and $B_{10}$ from level 4 are examined. $(B_9 \rightarrow B_{11})$ is a J-edge (see Figure 4.5(b)), thus $DF^+ = \{B_{11}\}$, and $PB = PB \cup \{B_{11}\}$. Node $B_{10}$'s only outgoing J-edge is with node $B_{11}$, which is already in $DF^+$. Now we process node $B_{11}$ in level 4. $(B_{11} \rightarrow B_7)$ is a J-edge, therefore $DF^+ = \{B_{11}, B_7\}$, and $PB = PB \cup \{B_7\}$. We continue processing at level 2 with node $B_4$. $(B_4 \rightarrow B_6)$ is a J-edge, hence $DF^+ = \{B_{11}, B_7, B_6\}$ and $PB = PB \cup \{B_6\}$. The only other J-edges in the DJ-graph are directed to node $B_6$, thus we are done, and $\phi$-functions for $x$ can be added to nodes $B_6$, $B_7$, and $B_{11}$. This process can be repeated with other initial sets; in particular, with the sets representing assignments of the other variables in the CFG.

### 4.1.3 Bilardi and Pingali

The third $\phi$-placement algorithm was first introduced by Bilardi and Pingali in 1995 [28]. In 2003, they revisited the description of the original algorithm in an extensive comparative study of SSA construction techniques [5, 6]. This algorithm uses a

[1] $PB$ refers to the "PiggyBank" used in Sreedhar and Gao's algorithm [32].

new data structure, the *augmented dominator tree.* First, the dominance frontier is defined in terms of edges instead of nodes.

**Definition 24.** Suppose $(B_i \rightarrow B_j)$ is an edge in a $CFG = G(V, E)$. If $B_i \neq idom(B_j)$, then the edge $(B_i \rightarrow B_j)$ is an *up-edge* [5].

**Definition 25.** An edge $(B_i \rightarrow B_j)$ is in the *edge dominance frontier (EDF)* of a node $B_k$ if $B_k$ dominates $B_i$ and $B_k$ does not strictly dominate $B_j$ [5].

In the Cytron *et al.* definition of a dominance frontier given in Definition 15, the node $B_j$ would be in the dominance frontier of $B_k$.

**Definition 26.** Let $B_k$ be a node in a $CFG = G(V, E)$ such that an edge $(B_i \rightarrow B_j) \in EDF(B_k)$. Then, $B_j \in DF(B_k)$ [5].

Finally, we need to know which nodes are *boundary nodes.* Several ways of determining boundary nodes were discussed in [5]. For example, a simple problem formulation defined a node to be a boundary node if it is a leaf node in the dominator tree. It was also suggested that every node could be initialized as a boundary node. In practice, however, boundary nodes are defined by Definition 28.

**Definition 27.** A *zone* is a smaller tree created by partitioning the dominator tree. The zone associated with a tree node $B_i$ is denoted $Z_{B_i}$. The *zone size* of $Z_{B_i}$ is $z[B_i]$ [5].

**Definition 28.** A node $B_i$ is a *boundary node* if:

*1.* $B_i$ is a leaf node in the dominator tree; *or*

*2.* $(1 + \sum_{B_j \in children(B_i)} z[B_j]) > (\beta \times |EDF(B_i)| + 1)$,

where $\beta \geq 0$ is a parameter used to control the space and query-time tradeoffs [5].[2]

**Definition 29.** A node $B_i$ in the dominator tree is an *interior node* if $B_i$ is not a boundary node [5].

**Definition 30.** The zone size of a node $B_i$ is computed by the following [5]:

$$z[B_i] = \begin{cases} 1 + \sum_{B_j \in children(B_i)} z[B_j], & \text{if } B_i \text{ is an interior node.} \\ 1, & \text{if } B_i \text{ is a boundary node.} \end{cases} \tag{4.3}$$

[2]For the remainder of this discussion, it can be assumed that $\beta = 1$. This $\beta$ value produced the best results in the experiments of [5], and was encouraged for use by the authors.

**Definition 31.** The *augmented dominator tree (ADT)* consists of a number of structures [5]:

1. a dominator tree capable of producing top-down and bottom-up traversals.
2. the depth-first search number (equivalently, the level number, discussed in Section 4.1.2) for each node of the tree.
3. a boolean value for each node indicating whether or not it is a boundary node.
4. a list of CFG edges corresponding to a particular node $B_i$. The list is $EDF(B_i)$ if $B_i$ is a boundary node. If $B_i$ is an interior node, the list consists of the CFG up-edges sourced at $B_i$.

The *ADT* for the example in Figure 4.1 is given in Table 4.1.

| Node | Level | Boundary Node? | List |
|---|---|---|---|
| $B_1$ | 0 | T | 6, 6, 6 |
| $B_2$ | 1 | T | 6 |
| $B_3$ | 1 | F | |
| $B_4$ | 2 | T | 6 |
| $B_5$ | 2 | T | 6 |
| $B_6$ | 1 | T | 7 |
| $B_7$ | 2 | T | 7 |
| $B_8$ | 3 | T | 7, 11, 11 |
| $B_9$ | 4 | T | 11 |
| $B_{10}$ | 4 | T | 11 |
| $B_{11}$ | 4 | T | 7 |
| $B_{12}$ | 3 | T | |

Table 4.1: The *ADT* for Figure 4.1

The algorithm, based on the *ADT* data structure, takes as input a set of nodes and returns the set of *merge nodes* where $\phi$-functions are to be placed.

**Definition 32.** Suppose $B_0$ is the start node of a $CFG = G(V, E)$. The *merge* relation ($M$) is a binary relation on nodes defined as a set of pairs, $(B_i, B_j)$ such that $B_i \in V$ and $B_j \in J(\{B_0, B_j\})$. The *merge set* of $B_i$ is the set of all nodes $B_j$ such that $(B_j, B_i) \in M$ [5].

The relationship between merge nodes and $\phi$-functions is given in Theorem 2.

**Theorem 2.** *The iterated dominance frontier is the same as the merge relation. That is* [5],

$$DF^+ = M \tag{4.4}$$

The input set of nodes are initialized as a priority queue, $PQ$, using the node levels as keys. The dominator tree is also required, and all nodes from the input set are marked in the tree. The main loop of the algorithm iterates while $PQ$ is not empty, taking the next deepest node $B_i$ from $PQ$ for processing. Then each node from the list for $B_i$ described in Definition 31, part 4, is studied. If the immediate dominator of the current list node is a strict ancestor of $B_i$, then it is a merge node, and is added to $M$. If the node was not marked in the dominator tree, it is marked and inserted in $PQ$ for future processing. Finally, if $B_i$ is not a boundary node, then we recursively visit all children of $B_i$ that aren't marked.

Consider the example in Figure 4.1. Let the input set $S = A(x) = \{B_2, B_4, B_5, B_9, B_{10}\}$. Also, $PQ = S = \{B_2, B_4, B_5, B_9, B_{10}\}$. Using the level as the key to $PQ$ means nodes $B_9$ and $B_{10}$ are processed first. The list for node $B_9$ consists just of node $B_{11}$. From Figure 4.5(a), we know that the immediate dominator of node $B_{11}$ is node $B_8$, which is a strict ancestor of node $B_9$. Thus, node $B_{11}$ is a merge node and $M = M \cup \{B_{11}\}$. Node $B_{11}$ is not in $S$, thus it is not marked in the dominator tree. It is then marked and $PQ = PQ \cup \{B_{11}\}$. The *ADT* in Table 4.1 shows that node $B_9$ is a boundary node, therefore this loop iteration is finished. We next process node $B_{10}$, whose only list element is node $B_{11}$, which is already in $M$. Then node $B_{11}$ itself is examined, since its level is also 4. Node $B_{11}$'s list consists of node $B_7$, whose immediate dominator is node $B_6$, a strict ancestor of node $B_{11}$. Thus, $M = \{B_{11}, B_7\}$. Node $B_7$ is not in $S$, hence it is marked in the dominator tree and $PQ = PQ \cup \{B_7\}$. Node $B_{11}$ is also a boundary node. Node $B_4$, with the next highest level number, is then extracted from $PQ$. Node $B_6$ is the only node in node $B_4$'s list. Node $B_6$'s immediate dominator is node $B_1$, a strict ancestor of all other nodes. Node $B_6$ is a merge node and $M = \{B_{11}, B_7, B_6\}$. Node $B_6$ is not in $S$, thus it is marked and added to $PQ$. And node $B_4$ is a boundary node. From Table 4.1, we can see that all the lists of the remaining nodes contain nodes that have already been added to $M$, therefore we are done. $\phi$-functions for $x$ can then be added to nodes $B_6$, $B_7$, and $B_{11}$. To obtain the required $\phi$-functions for other variables, other input sets can be used; namely, $A(v)$ for all other $v$.

## 4.2 Variable Renaming

The renaming of variables that subsequently occurs during SSA construction is much less studied in the literature. Cytron *et al.* propose renaming variables using a top-down traversal of the dominator tree [19]. During this pass, arrays of stacks are accessed to find the next available variable for an assignment, or the previous definition that should now be referenced. The array is indexed by the original variable name, and the stacks contain the replacement subscripts. By visiting a specific node, statements associated with that node, beginning with any $\phi$-functions, are processed in sequential order. Only variables referenced in the statement are handled.

Briggs *et al.* presented improvements to this algorithm in 1998 [7]. They proposed pushing a subscript on the stack only at the first definition of a variable $x$ in the block. Then, subsequent definitions would overwrite the subscript, thus taking away the pure stack nature of the data structure. Information would have to be maintained as to which variables had subscripts pushed into a particular block. To restore the original state of the stack, the algorithm reads the already-pushed list, and pops subscripts from that list.

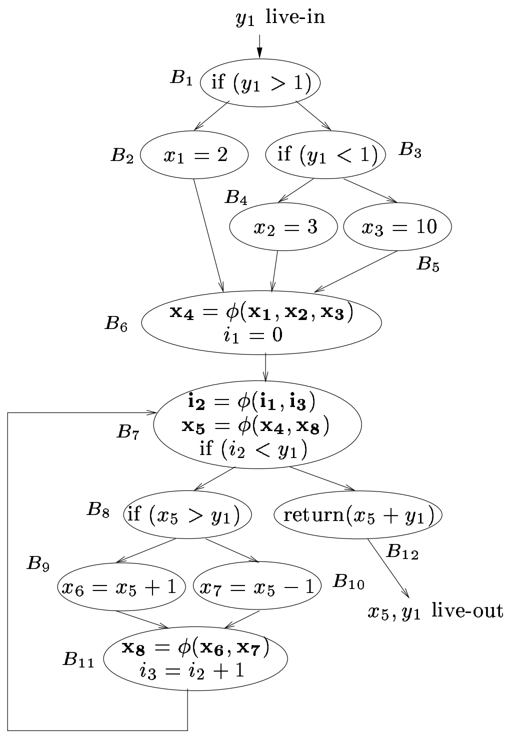


Figure 4.6: The SSA form of Figure 4.1

The SSA form of our running example is given in Figure 4.6. This product results from any of the techniques discussed in Sections 4.1 and 4.2. Added $\phi$-instructions are shown.

## 4.3 Discussion

Cytron *et al.*'s algorithm from Section 4.1.1 is widely thought to be the easiest of the three presented algorithms to implement. It is thus still found in many production compilers today. Since the algorithm is based on the dominance frontiers of individual nodes, the calculation of these structures is crucial in compile-time analysis. Consider a CFG with $N$ nodes, $E$ edges, $A_{tot}$ number of assignments and $M_{tot}$ number of references to variables. Let $DF$ be the mapping from nodes to their dominance frontiers,[3] and $avg(DF)$ be the weighted average of the sizes $|DF(B)|$. $X$ is the set of all nodes in the CFG. Then the total running time of the algorithm is the time required to compute the dominance frontiers, along with the time to place $\phi$-functions and the time to rename variables [19]. That is,

$$Time = O\left(\sum_{B \in X} |DF(N)|\right) + O(A_{tot} \times avg(DF)) + O(M_{tot}) \tag{4.5}$$

Now let $T$ be the overall size of the original program, calculated by:

$$T = max\{N, E, A_{orig}, M_{orig}\} \tag{4.6}$$

The worst-case running time is then [19]:

$$Time_{worst} = O(R^2) + O(R^3) \tag{4.7}$$

The authors argue that in practice, the calculation is actually linear.

Sreedhar and Gao's DJ-graph algorithm for placing $\phi$-functions presented in Section 4.1.2 is in fact linear. Given a dominator tree, the DJ-graph can be constructed in $O(E)$ time, just the time required to insert the J-edges. It was shown that [32]:

**Theorem 3.** *The time complexity of Sreedhar and Gao's algorithm is* $O(|E|)$.

Now let $V$ be the number of variables in the CFG. Sreedhar and Gao's method takes as an initial set the set of assignments to a particular variable. The algorithm will have to be performed for each variable in the CFG to produce the complete

[3]Assume for all of these calculations that the dominator tree is available. It has been shown that the dominator tree computation is $O(E)$ [23, 22].

set of iterated dominance frontiers. Therefore, the actual time to compute the SSA form of a program using this method is:

$$Time = O(E) + |V| \times O(|E|) + O(M_{tot}) \tag{4.8}$$

The final algorithm presented in Section 4.1.3 was that of Bilardi and Pingali, which places $\phi$-functions based on another new data structure, the $ADT$. Let $E_{up}$ be the set of up-edges in the CFG. It was shown that [5]:

**Theorem 4.** *The ADT for a given CFG can be constructed in time*

$$Time_{ADT} = O(|E_{up}| + (1 + 1/\beta) \times |N|) \tag{4.9}$$

Given that the version of the algorithm presented here uses $\beta = 1$, Equation 4.9 becomes:

$$Time_{ADT} = O(|E_{up}| + 2|N|) \tag{4.10}$$

Let $F$ be the number of extractions from the $PQ$ data structure and $K$ be the number of keys used by the $PQ$ implemented as a heap. Then the $\phi$-function placement algorithm was shown to be [5]:

$$Time_{\phi-function} = O(|N| + |E_{up}| + |V|/c) + O(F + K), \ \ c \text{ constant} \tag{4.11}$$

The final running time of Bilardi and Pingali's method is then:

$$Time = O(|E_{up}| + 2|N|) + O(|N| + |E_{up}| + |V|/c) + O(F + K) + O(M_{tot}), \ \ c \text{ constant} \tag{4.12}$$

From the experience of producing the examples seen throughout Section 4.1, it was easy to rank the algorithms in practice. Sreedhar and Gao's method was the easiest to work through, since the DJ-graph made visualizing the process straightforward. Cytron *et al.*'s algorithm was easy to understand, since it is rooted in original theory without additional structures to learn. It simply requires the basic concepts that the other algorithms use only as a starting point. It is also the most familiar, as it appears in most compilers. However, the enhancement of a concrete data structure could be beneficial. Bilardi and Pingali's technique was very complicated. There were quite a few structures and values to keep track of, and this process was tedious. The idea was not intuitive.

Since the dominator tree is readily available, Cytron *et al.*'s algorithm is still probably the easiest to implement, as no new data structure is required. However,

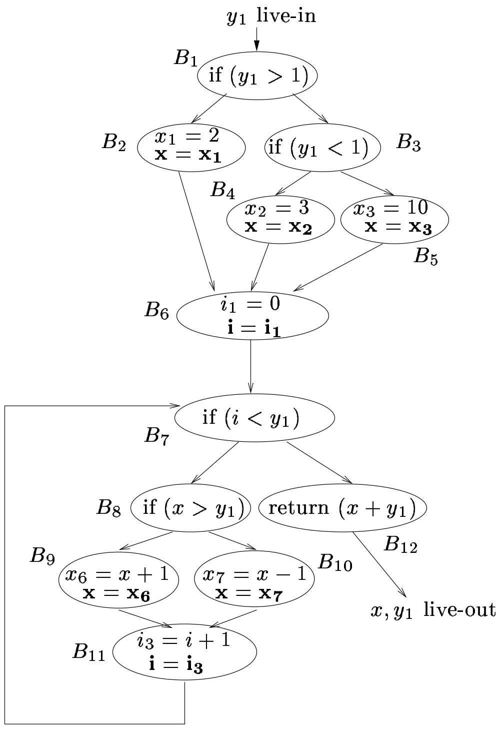


Figure 4.7: The result of naively inserting copies to remove the SSA form

constructing a DJ-graph from the dominator tree should require little extra time. The main concern therefore with Sreedhar and Gao's method would be the additional space requirements for a structure only required for one purpose. Bilardi and Pingali's method also needs an additional structure for this single task, but the cost of building the elaborate *ADT* is not worth the supposed rewards.

## 4.4 Conversion out of SSA

After SSA translation, a code representation ensues that contains non-executable instructions. Further compilation phases, such as instruction scheduling and register allocation, require the removal of these $\phi$-functions.

Trivially, the removal process can be achieved by inserting many copy instructions into the modified code, one for each definition of a variable. Consider Figure 4.7. In our example, five copies for $x$ are inserted, one in each of nodes $B_2, B_4, B_5, B_9$ and $B_{10}$. These ensure that when the common uses of $x$ occur in nodes $B_8, B_9, B_{10}$ and $B_{12}$, the correct definition will be used. Then the $\phi$-instructions in nodes $B_6, B_7$ and $B_{11}$ are no longer necessary, and are removed. A similar process is used for $i$. The added instructions are indicated in bold print.

The number of copies will increase according to the original code size and complexity. Methods for minimum copy insertion are desired.

Sreedhar *et al.* have proposed three methods for translating out of the SSA form [33]. These methods use a variety of techniques, ranging from brute-force insertion of copies to using both dataflow and *interference graph* information.

**Definition 33.** Consider a $CFG = G(V, E)$ such that $x$ and $y$ are variables in $G$. An *interference graph (I)* can be used to indicate if $x$ and $y$ interfere. Let each variable in $G$ represent a node in $I$. If $x$ and $y$ interfere, then there is an edge between the nodes representing $x$ and $y$ in $I$.

### 4.4.1 Naive Translation

Using this method, copies are inserted for all variables *referenced* in a $\phi$-instruction.

**Definition 34.** Given a $\phi$-instruction of the form $x = \phi(x_1, x_2, \ldots, x_n)$, $x$ and all the source operands $x_1, \ldots, x_n$ are said to be *referenced* in that instruction.

Contrary to the preliminary example presented in Figure 4.7, this technique will also add copies for the target of the $\phi$-instruction. The result of applying the Naive Translation Method to our running example can be seen in Figure 4.8. Added copies are indicated in bold print.

Let's investigate the $\phi$-instructions for $x$ in Figure 4.6. Since there are three instructions, three copies of the form $x_i = x$ will be required for each target operand $x_i$. Similarly, there are seven $\phi$-function source operands $x_j$ that require copies of the form $x = x_j$. These copies ensure that the correct value is accessed during uses of $x$. However, the ten copies for a single variable $x$ seems excessive. Improvements are needed.

In production compilers, however, SSA will never be translated into and out of directly without its benefits being maximized. In this regard, we can assume that several optimizations will occur between the SSA form construction and SSA removal. Sreedhar *et al.*'s remaining translation methods are best seen in this altered context. For this discussion, we will focus on a slightly modified example, shown in Figure 4.9. Here, some instructions have been re-arranged, and some basic blocks removed, as can happen after compiler optimizations.

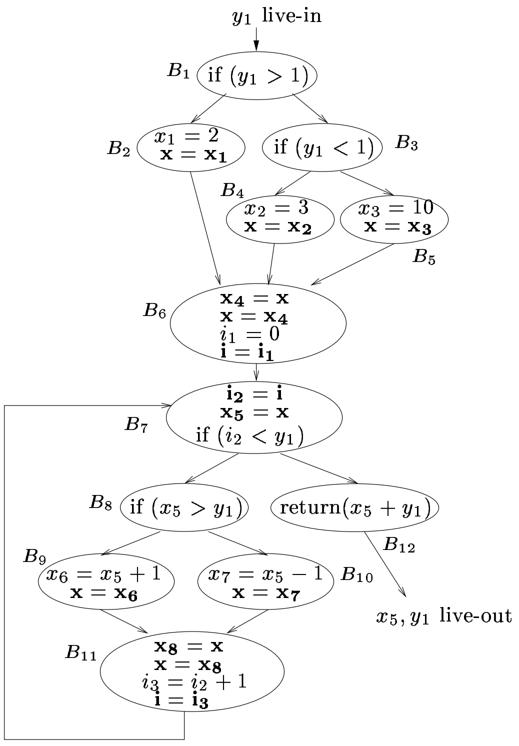


Figure 4.8: Translating out of SSA using Sreedhar *et al.*'s first method

### 4.4.2 Translation Based on Interference Graph Update

This translation method sees the insertion of copies only for variables whose live ranges interfere. Cytron and Gershbein present a special definition of liveness with respect to $\phi$-instructions [20]. In particular, if a $\phi$-function $x_\phi$ is in basic block $B_j$, then each use of a $\phi$-function source operand $x_i$ is associated with the end of the corresponding predecessor to $B_j$ through which $x_i$ reaches $B_j$. $\phi$-functions are expected to occur at the beginning of the basic block in which they appear. Given these assumptions, the subsequent definition follows.

**Definition 35.** Consider a source operand $x_i$ of a $\phi$-instruction $x_\phi$ that occurs in basic block $B_j$. Let basic block $B_i$ be the block through which $x_i$ reaches $x_\phi$. $x_i$ is *live* along the path from the point right after its definition to the final point in $B_i$. The $\phi$-instruction target $x$ is *live* along the path from the point right after its definition to the point right before its last use.

Given Definition 35, target operands of a $\phi$-function cannot be live at the same time as the source operands of that $\phi$-function. We can thus eliminate all of the copies related to $\phi$-function target operands from Figure 4.8. The result of removing

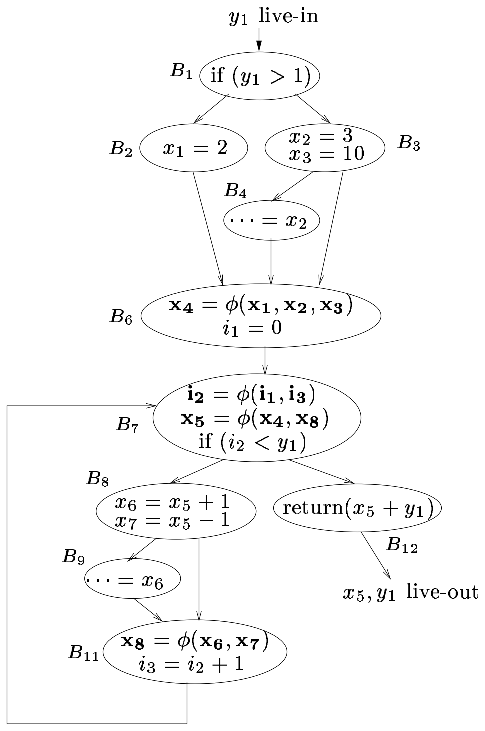


Figure 4.9: Modified example

the SSA form from Figure 4.9 is given in Figure 4.10.

Let's study the $x$ $\phi$-instructions from Figure 4.9. We must determine which referenced variables from $\phi$-functions interfere. The three copies required for the three $\phi$-function targets are automatically eliminated. Since only $x_2$ and $x_3$ of the operands for the $\phi$-function defining $x_4$ interfere, the copy for $x_1$ is unnecessary. The main $x$ variable is then propagated throughout the code, and we are left with four copies for $x$.

This result is a significant improvement for a small example. Of course, there is a cost for examining each variable's live range, but with such benefits, it is worth the additional checks. Section 4.4.3 analyzes the effects of further steps.

### 4.4.3 Translation Based on Data Flow and Interference Graph Updates

With this method, copies are inserted based on live ranges that interfere, and the live-in and live-out sets of the variables involved. Eliminating interferences between $\phi$-instruction source operands can be done exclusively with live-out sets, while eliminating those between target and source operands requires the live-in sets as well. It is the most effective of Sreedhar *et al.*'s methods.

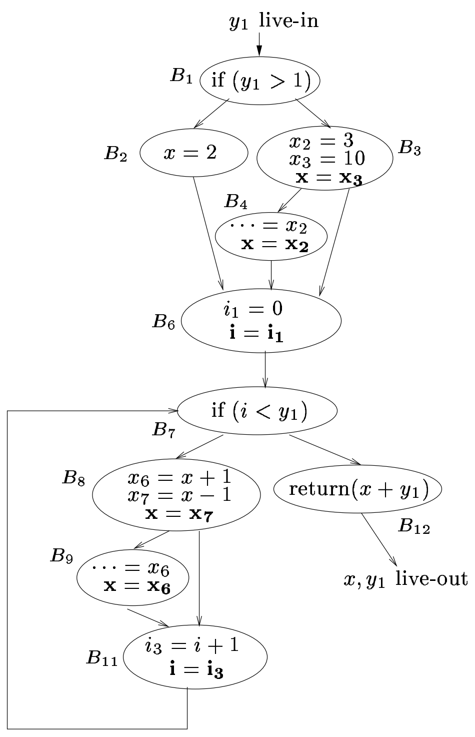


Figure 4.10: Translating out of SSA based on live range interference

**Definition 36.** Two variables $x$ and $y$ are in the same *$\phi$-congruence class* (denoted by $\phi_{cc}$) if they are referenced in the same $\phi$-instruction, or there exists a resource $z$ such that $y$ and $z$ are referenced in the same $\phi$-instruction, and $x$ and $z$ are referenced in the same $\phi$-instruction [33].

Intuitively, two variables are congruent if they are referenced in the same $\phi$-instruction, or referenced transitively between $\phi$-instructions. We want two congruent variables to be able to receive the same representative name. It can be compared to register allocation by colouring, where variables that do not interfere can be assigned to the same register [11]. Similarly, if two variables occur in the same $\phi$-congruence class, we would like for them to get renamed to the same representative name upon removal of the SSA form.

A fundamental property for this translation is the *$\phi$-Congruence Property*, a slight modification on which is presented here [33].

**Property 1.** *($\phi$-Congruence Property)* All occurrences of variables that belong to the same $\phi$-congruence class in a program may be replaced by the same representative name. After all variables in the $\phi$-instruction have been replaced, the $\phi$-instruction can be eliminated without violating the original semantics of the program, and thus the SSA form can be removed.

As Property 1 states, congruent variables may be renamed the same, but this renaming is not always possible. Interfering variables within a $\phi$-congruence class should be handled by the insertion of copies. In fact, there are four cases to be studied with respect to the $\phi$-congruence classes.[4] A $\phi$-congruence class is initialized so that a variable in a $\phi$-instruction belongs to its own class; hence, $\phi_{cc}(x_i) = \{x_i\}$. Given a variable $x_i$, the basic block through which its definition reaches the $\phi$-instruction is denoted $B_{x_i}$.

1. If $[\phi_{cc}(x_i) \cap L_{out}(B_{x_j}) \neq \emptyset] \wedge [\phi_{cc}(x_j) \cap L_{out}(B_{x_i}) = \emptyset]$, then the copy $x_{i'} = x_i$ is needed in $B_{x_i}$. This copy ensures $x_i$ and $x_j$ are added to different $\phi$-congruence classes.

2. If $[\phi_{cc}(x_i) \cap L_{out}(B_{x_j}) = \emptyset] \wedge [\phi_{cc}(x_j) \cap L_{out}(B_{x_i}) \neq \emptyset]$, then the copy $x_{j'} = x_j$ is needed in $B_{x_j}$.

3. If $[\phi_{cc}(x_i) \cap L_{out}(B_{x_j}) \neq \emptyset] \wedge [\phi_{cc}(x_j) \cap L_{out}(B_{x_i}) \neq \emptyset]$, then two copies, $x_{i'} = x_i$ in $B_{x_i}$ and $x_{j'} = x_j$ in $B_{x_j}$.

4. If $[\phi_{cc}(x_i) \cap L_{out}(B_{x_j}) = \emptyset] \wedge [\phi_{cc}(x_j) \cap L_{out}(B_{x_i}) = \emptyset]$, then one of the copies $x_{i'} = x_i$ in $B_{x_i}$ or $x_{j'} = x_j$ in $B_{x_j}$ is needed. The final decision is made at a later stage of the translation process.

When all required copies have been added, the variables denoted by $x_{i'}$ can be replaced by a representative name.

Using these conditions, the translated code in Figure 4.11 is produced. Compared with Figure 4.10, we have eliminated two copies, the copy in node $B_3$ and the one in node $B_8$. Consider the variables $x_2$ and $x_3$ from Figure 4.9, where $B_{x_2} = B_4$ and $B_{x_3} = B_3$. We know that $\phi_{cc}(x_2) = \{x_2\}$ and $\phi_{cc}(x_3) = \{x_3\}$. We can also determine that $L_{out}(B_{x_2}) = L_{out}(B_4) = \emptyset$ and $L_{out}(B_{x_3}) = L_{out}(B_3) = \{x_2\}$. Given the first case described previously, we check $\phi_{cc}(x_2)$ versus $L_{out}(B_{x_3})$. That is, $\phi_{cc}(x_2) \cap L_{out}(B_{x_3}) = \{x_2\} \cap \{x_2\} = \{x_2\} \neq \emptyset$. We also look at $\phi_{cc}(x_3)$ and $L_{out}(B_{x_2})$. Thus, $\phi_{cc}(x_3) \cap L_{out}(B_{x_2}) = \{x_3\} \cap \emptyset = \emptyset$. Hence, variables $x_2$ and $x_3$ satisfy the first case of the four to be checked, and only the copy $x_{2'} = x_2$ is needed in $B_{x_2}$. The copy in $B_3$ is therefore unnecessary, and not included. Upon renaming

[4]Budimlić *et al.* discussed a similar approach in [8], where variables are compared for interference by checking the liveness information for the blocks in which the respective variables are defined.

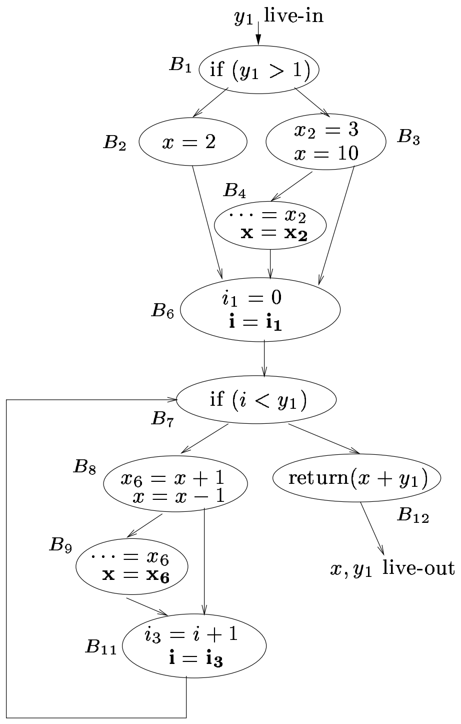


Figure 4.11: Translating out of SSA based on live range interference and dataflow information

by the representative name, the actual copy $x = x_2$ is added in node $B_4$. A similar analysis can be performed for the variables $x_6$ and $x_7$ in nodes $B_8$ and $B_9$ to see that only the copy $x = x_6$ is required in $B_9$, and the copy from $B_8$ in Figure 4.10 is unnecessary.

### 4.4.4 Comparison of Individual Translation Methods

As can be seen by the examples presented in Sections 4.4.1, 4.4.2 and 4.4.3, Sreedhar *et al.*'s three translation techniques produce a variety of results, even on small pieces of code. Even though the results are all correct, some are more desirable than others. In particular, producing fewer inserted copies will result in a smaller number of extra instructions to be executed, and an ultimate decrease in additional run-time. However, an overhead is incurred during the increased work performed by translation methods 2 and 3. In Chapter 8, the actual costs and benefits of methods 1 and 2 will be examined.

In terms of working with the individual translation methods, there are clear differences in usability. The naive method is straightforward, as copies are inserted exclusively for variables referenced in $\phi$-instructions. Working with this method simply entails iterating through the $\phi$-instructions to see which variables are referenced.

A little more thought is required to examine the live ranges of variables as performed in the second translation method. Since the liveness information has already been computed at an earlier stage of the compilation process, the additional work needed is minimal. It is simply a matter of maintaining and updating the liveness information throughout the SSA construction and removal phase. The benefits seem obvious, since even in our small example more than half of the copies included using the first method could be eliminated by exploring liveness properties.

The third translation method, however, requires extra processing which at present is not needed by other compilation phases. The calculations necessary to implement $\phi$-congruence classes may not be worth the additional effort. Besides analyzing the liveness information, the $\phi$-congruence classes must also be compared to the live sets. It is not yet clear if the added compile-time restrictions will be alleviated by significant runtime benefits. However, as was seen by the small example of Section 4.4.3, the major gains were realized between methods 1 and 2, and much smaller improvements were achieved through method 3.

# Chapter 5

# SSA for Predicated Code

## 5.1 Predication

Traditional SSA only applies to code with branches by "choosing" the path that the program execution followed. However, the technique of *if-conversion*, introduced by Allen *et al.* in 1983 [2], eliminates conditional branches and changes the flow of a program [3]. It enables the compiler to treat control dependences as data dependences. If-converted code is sequential, but removing control flow cannot change the semantics of the program. With conditional branches removed, decisions are made based on *predicates* [29, 27]. Each statement is assigned a logical expression, that if evaluated true results in the statement being executed. The predicates themselves are defined by statements inserted in the program. If a program statement has no explicit predicate, the predicate is assumed to be true, and the statement is always executed.[1]

Recall the example from Chapter 4 in Figure 5.1(a). There are three occurrences of conditional branches associated with if-statements. By performing if-conversion, these branches will be eliminated. Figure 5.1(b) gives the if-converted form.

SSA as it has been defined does not deal with predicated code. The transformation has no way of "deciding" which statements are executed based on the predicate information. Consider Figure 5.1(b). There are three predicated assignments to $x$ before $x$ is used. After applying the SSA algorithm to this code, there is still no decision as to which value of $x$ to use. Traditional SSA is not sufficient to deal with this situation, since many definitions of a variable can still reach an use in a single control-flow path [36].

[1]For this discussion, the predicate $p_0$ will be the always true predicate, thus statements assigned to $p_0$ will always be executed.

```
$y$ live-in
if $(y > 1)$
    $x = 2$
else if $(y < 1)$
    $x = 3$
    else $x = 10$
for $(i = 0; i < y; i++)${
    if $(x > y)$
      $x++$
    else
      $x--$
}
return$(x + y)$
$x, y$ are live-out
```

(a)

```
$y$ live-in
$(p_0)$  $p_1, p_2 = (y > 1)$
$(p_1)$      $x = 2$
$(p_2)$  $p_3, p_4 = (y < 1)$
$(p_3)$      $x = 3$
$(p_4)$      $x = 10$
$(p_0)$      $i = 0$
label:
$(p_0)$  $p_5, p_6 = (i < y)$
$(p_5)$  $p_7, p_8 = (x > y)$
$(p_7)$      $x++$
$(p_8)$      $x--$
$(p_5)$      $i++$
$(p_5)$  br: label
$(p_6)$  return$(x + y)$
$x, y$ are live-out
```

(b)

Figure 5.1: (a) Example from Figure 4.1; (b) If-converted example

It is not desirable for SSA to ignore if-converted code. If-conversion is a popular and useful optimization technique since branches can hinder most compiler analyses. Current production compilers translate out of SSA form well before if-conversion occurs to avoid the problem. However, it is a natural extension to want code in SSA as long as possible within intermediate representations, to maximize the benefits SSA can afford.

## 5.2 $\psi$-SSA

Stoutchinin and de Ferriere introduced an SSA algorithm for predicated code in 2001 [36]. They suggested that their technique could benefit Linear Assembly Optimizers and just-in-time compilers,[2] as well as managing inlined predicated assembly code in higher level programs. These advantages are especially prevalent in architectures with support for predication, such as the target architecture for this work, the Intel Itanium processor [16].

The algorithm presented in [36], called $\psi$-SSA, is an extension of the traditional

[2]Linear Assembly Optimizers take programs written in a linear assembly input language and translate it into the traditional assembly language used at assembly and linkage-time [36]. Just-in-time compilers convert Java bytecodes into executable instructions.

```
$y_1$ live-in
($p_0$)  $p_1, p_2 = (y_1 > 1)$
($p_1$)      $x_1 = 2$
($p_0$)      $x_2 = \psi(x_1)$
($p_2$)  $p_3, p_4 = (y_1 < 1)$
($p_3$)      $x_3 = 3$
($p_0$)      $x_4 = \psi(x_2, x_3)$
($p_4$)      $x_5 = 10$
($p_0$)      $x_6 = \psi(x_4, x_5)$
($p_0$)      $i_1 = 0$
label:
($p_0$)      $i_2 = \phi(i_1, i_3)$
($p_0$)  $p_5, p_6 = (i_2 < y_1)$
($p_0$)      $x_7 = \phi(x_6, x_{11})$
($p_5$)  $p_7, p_8 = (x_7 > y_1)$
($p_7$)      $x_8 = x_7 + 1$
($p_0$)      $x_9 = \psi(x_7, x_8)$
($p_8$)      $x_{10} = x_9 - 1$
($p_0$)      $x_{11} = \psi(x_9, x_{10})$
($p_5$)      $i_3 = i_2 + 1$
($p_5$)  br: label
($p_6$)  return($x_{11} + y_1$)
$x_{11}, y_1$ are live-out
```

Figure 5.2: $\psi$-converted form of Figure 5.1(b)

SSA representation. The technique inserts $\psi$-functions at predicate join points, similar to the $\phi$-function insertion of SSA. The operands of the $\psi$-function represent the predicated definitions that reach a particular program point. The algorithm first inserts $\psi$-functions after conditional assignments to variables, and then performs the entire SSA procedure, including $\phi$-function placement.

Figure 5.2 gives the code of Figure 5.1(a) in SSA form, including the transformation into $\psi$-SSA. Notice in Figure 5.2 that there are two fewer $\phi$-functions than in the SSA code of Figure 4.6. Since $\psi$-SSA is applied after if-conversion, many of the $\psi$-functions are simply if-converted $\phi$-functions.

As with any SSA transformation, reverting the code back into an executable form is necessary. This translation is usually non-trivial since an assortment of optimizations may have been performed by this stage in the compiler. Similar to the naive method of translation out of the SSA form presented in Section 4.4.1, a predicated copy instruction could be inserted for every $\psi$-function operand. This translation technique could result in excess copies being inserted.

As part of the work in [36], a translation algorithm was presented to remove the $\psi$-SSA form. This method makes associations between related predicated assignments and creates a representative live range for the related assignments. The fundamental idea behind the algorithm is that of a *$\psi$-congruence class.*

**Definition 37.** Two variables $x$ and $y$ belong to the same *$\psi$-congruence class* (and are said to be congruent to each other) if they are referenced in the same $\psi$-function, or there exists a variable $z$ such that $x$ is congruent to $z$ and $z$ is congruent to $y$ [36].

Definition 37 is closely related to Definition 36 presented in Section 4.4.3. We want to replace all variables in the same $\psi$-congruence class with a single representative name upon translation out of the $\psi$-SSA form. Let $x_i$ be an element of a $\psi$-congruence class. $x_i$ actually corresponds to the live subrange beginning at $x_i$'s predicated definition and ending with $x_i$'s last use not in a $\psi$-function. The renaming is then possible since each $\psi$-congruence class represents a single live range, the union of the non-overlapping subranges.

The *congruence class order,* $\prec_c$, is used to relate elements in a single $\psi$-congruence class, and help maintain the original program semantics.

**Definition 38.** Given two variables $x$ and $y$, $x \prec_c y$ if [36]:

1. the definitions of $x$ and $y$ may be live at the same time; *and*

2. $x$ precedes $y$ in the operand list of some $\psi$-function, or there exists a variable $z$ such that $x$ precedes $z$ in the operand list of some $\psi$-function, and $y$ and $z$ are referenced in some $\psi$-function with $z \prec_c y$.

The $\psi$-SSA form must maintain a certain consistency, defined by the following conditions [36]:

1. Assignments to variables within each $\psi$-congruence class must occur in the congruence class order.

2. Live subranges corresponding to elements of each $\psi$-congruence class can not interfere.

Often, transformations such as code motion may result in a non-consistent $\psi$-SSA form. In this situation, copy instructions must be inserted to restore the code's

```
$y$ live-in
($p_0$)   $p_1, p_2 = (y_1 > 1)$
($p_1$)       $x = 2$
($p_2$)   $p_3, p_4 = (y_1 < 1)$
($p_3$)       $x = 3$
($p_4$)       $x = 10$
($p_0$)       $x_6 = x$
($p_0$)       $x = x_6$
($p_0$)       $i_1 = 0$
($p_0$)       $i = i_1$
label:
($p_0$)   $p_5, p_6 = (i < y_1)$
($p_0$)       $x_7 = x$
($p_5$)   $p_7, p_8 = (x_7 > y_1)$
($p_7$)       $x = x_7 + 1$
($p_0$)       $x_9 = x$
($p_8$)       $x = x_9 - 1$
($p_0$)       $x_{11} = x$
($p_0$)       $x = x_{11}$
($p_5$)       $i_3 = i + 1$
($p_5$)       $i = i_3$
($p_5$)   br: label
($p_6$)   return($x_7 + y_1$)
$x_7, y_1$ are live-out
```

(a)

```
$y$ live-in
($p_0$)   $p_1, p_2 = (y_1 > 1)$
($p_1$)       $x = 2$
($p_2$)   $p_3, p_4 = (y_1 < 1)$
($p_3$)       $x = 3$
($p_4$)       $x = 10$
($p_0$)       $i_1 = 0$
($p_0$)       $i = i_1$
label:
($p_0$)   $p_5, p_6 = (i < y_1)$
($p_0$)       $x_7 = x$
($p_5$)   $p_7, p_8 = (x_7 > y_1)$
($p_7$)       $x = x_7 + 1$
($p_0$)       $x_9 = x$
($p_8$)       $x = x_9 - 1$
($p_5$)       $i_3 = i + 1$
($p_5$)       $i = i_3$
($p_5$)   br: label
($p_6$)   return($x_7 + y_1$)
$x_7, y_1$ are live-out
```

(b)

Figure 5.3: (a) After removal of SSA and $\psi$-SSA from Figure 5.2;
(b) Final code product after redundant copy removal

consistency. When the code is once again consistent, the renaming process can proceed. The result of translating the example of Figure 5.2 both out of $\psi$-SSA and SSA can be seen in Figure 5.3(a). Note that dead code has been removed at this stage as well. Figure 5.3(b) gives the final code product after eliminating redundant copies. The resultant code includes 4 extra instructions over the original if-converted form. This overhead can be justified by the additional optimization opportunities presented by the complete SSA form.

## 5.3 Predicated SSA

Carter *et al.* first introduced the notion of applying SSA to predicated code in 1999 [9, 10]. Their technique, Predicated SSA (PSSA), is designed for the *Trimaran*

*System* (Version 1.0) [38] and uses *hyperblocks* [24].

**Definition 39.** A *hyperblock* is a set of predicated basic blocks with one entry point at the beginning of the region, but one or more exit points from locations throughout the region.

A hyperblock consists of basic blocks, which are included in the hyperblock through profiling. Information about execution frequency, basic block size and operation latencies is compiled. A hyperblock should maximize optimization and scheduling opportunities by combining basic blocks of different control flow paths. Ideal blocks to include within a hyperblock are small and infrequently executed, with few hazardous instructions [24].[3] If branches in eligible basic blocks have both true and false targets within the hyperblock, the branches get if-converted. A property of the hyperblock is that it contains no cyclic dependences.

PSSA processes the hyperblock in top-down order, and takes two forms: Control PSSA, which is used on predicate-defining operations; and Normal PSSA, which applies to all other instructions. The algorithm introduces a new predicate OR operation, that defines predicates on blocks by taking the logical OR of multiple predicates. *Full-path predicates* are also used, along with path-sensitive analysis, to determine under which conditions an individual variable reached a join point [9].

**Definition 40.** A *full-path predicate* is a collection of predicates representing the unique path along which an operation is valid.

When processing the hyperblock, if an assignment is reached, Normal PSSA is invoked. The variable being defined is renamed and operands take on their already renamed versions. If the assignment is in a join block and multiple versions are live, the operation is duplicated in every incoming path with appropriate variable versions. Full-path predicates are used on these copies.

*Trimaran* defines a `cmpp` operation to assign values to predicates, and PSSA uses the instruction. Control PSSA is used to handle `cmpp` operations, by replacing them with one or more `cmpp` instructions that define full-path predicates for each path leading in to the block. The new `cmpp` instructions are guarded by the full-path predicate coming in to the current block.

[3]Hazardous instructions include procedure calls and memory accesses that are not readily resolvable.

The final step of the PSSA algorithm comes after optimizations (such as predicated speculation and control height reduction [9]) have been performed, when extra code is removed and copies are inserted to restore the code's consistency.

## 5.4 Comparison of $\psi$-SSA and PSSA

Both $\psi$-SSA and PSSA are attempting to remove conditional control flow from a program to enable potential optimization opportunities that were previously unseen. However, the methods employed to achieve this goal are quite different. For example, the architects of PSSA do not implement $\phi$-functions, claiming additional dependences would be added, thus making the hyperblock schedule longer. This exclusion results in instances of incomplete SSA code, leaving some SSA optimizations to falter. The main goal of PSSA is scheduling instructions at the earliest cycle, hence the loss of SSA potential is minor. Since $\psi$-SSA builds on the established SSA algorithm, maximum benefits can be achieved from both SSA and its predicated version.

The fundamental difference between PSSA and $\psi$-SSA is its usability. Since PSSA was implemented as part of the *Trimaran System*, which is a simulation system, adding the new predicate OR instruction was trivial. On a real architecture, such as the target IA-64, adding new instructions is not straightforward. A workaround is suggested in [9], which involves transferring the predicate register file into a general register with the `move from predicate` instruction provided in IA-64 [14]. But with no new instructions required by $\psi$-SSA, it is an easier method to implement.

# Chapter 6

# Open Research Compiler

## 6.1 Existing Functionality

The Open Research Compiler (ORC) [31] is an open source compiler project intended for leading research in compiler design and optimization. Based on the MIPSPro compiler, the project is headed by Intel [15] and the Chinese Academy of Sciences [26], and is geared towards Intel's Itanium architecture [16]. The compiler gives researchers and students an opportunity to test their ideas in a competitive environment.

The ORC currently implements SSA before the code generator, which is the case with all modern production compilers. The SSA form is removed before the backend begins to generate code. The code transformation is performed using Cytron *et al.*'s algorithm, presented in Section 4.1.1. Figure 6.1 shows the flow of control in the current ORC version.[1]

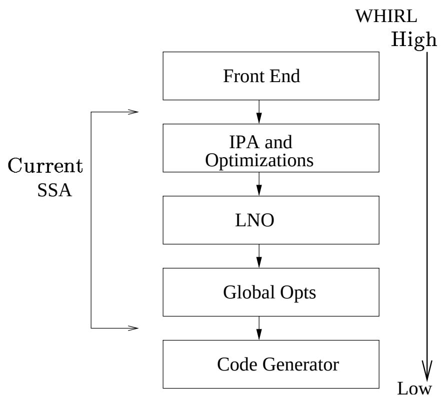


Figure 6.1: Flow of control in ORC

As Figure 6.1 indicates, the SSA transformation is performed immediately fol-

[1]The work in this thesis was performed on ORC version 2.0.

lowing the front end. Interprocedural analysis (IPA), interprocedural optimizations, loop nest optimizations (LNO), and global optimizations all occur within the SSA form. The intermediate transformation used by the ORC is WHIRL [37], and the SSA has been removed by the lowest phase of WHIRL. To this point, no attempt has been made to incorporate SSA in the code generator. The existing code generator is shown in Figure 6.2.

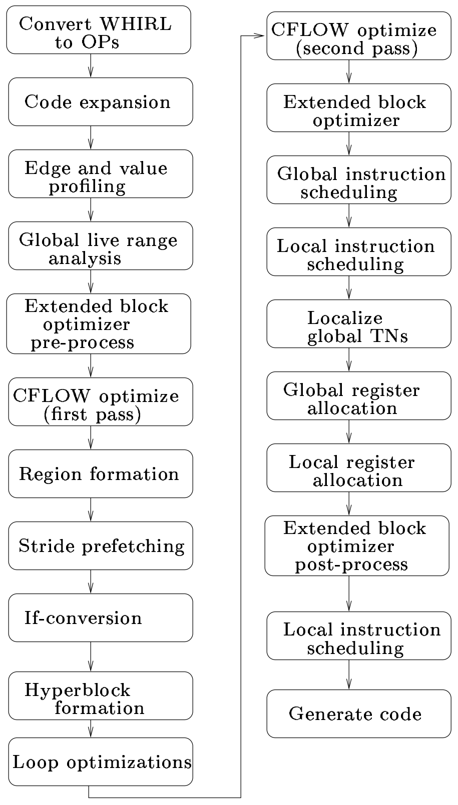

Convert WHIRL to OPs
Code expansion
Edge and value profiling
Global live range analysis
Extended block optimizer pre-process
CFLOW optimize (first pass)
Region formation
Stride prefetching
If-conversion
Hyperblock formation
Loop optimizations
CFLOW optimize (second pass)
Extended block optimizer
Global instruction scheduling
Local instruction scheduling
Localize global TNs
Global register allocation
Local register allocation
Extended block optimizer post-process
Local instruction scheduling
Generate code

Figure 6.2: ORC's code generator

The phases of the code generator are briefly described here.

**Convert WHIRL to OPs** The intermediate representation used up to this point, WHIRL, is removed in favour of actual operations.

**Code expansion** Among the tasks performed during this phase, BBs are split into smaller units and tail calls are optimized.[2]

**Edge and value profiling** This analysis is performed for profile-directed compilation, where information is gathered at runtime to aid further compilation

[2]A tail call is a recursive call that exists at the end of the recursive function, *i.e.*, there are no further instructions past the recursive call.

decisions. Edge profiling is the traditional method used to determine frequently executed paths in the CFG, while value profiling is used on individual variables to assist in value optimizations such as constant propagation.

**Global live range analysis** This pass actually occurs at several points following the current stage, to update the live ranges that may have been altered by individual transformations. Live ranges are determined at a global level, for use in many phases.

**Extended block optimizer pre-process** Used in the extended block optimization phase, blocks are analyzed here and transformed into an extended block sequence of instructions, beyond the existing basic block.

**CFLOW optimize (first pass)** Control flow based optimizations are performed, including unreachable code removal and branch optimization.

**Region formation** Within this step, a region is built on which to perform the subsequent stages. It is desirable to have the largest area possible to optimize, without creating an unmanageable chunk of code.

**Stride prefetching** This phase introduces a method for choosing candidates for software prefetching using information about strides between loop iterations.

**If-conversion** This stage removes conditional branches via predication, as discussed in Section 5.1.

**Hyperblock formation** During this pass, larger chunks of predicated basic blocks are fused for analysis, as in Definition 39.

**Loop optimizations** The traditional loop optimizations such as loop unrolling and backedge coalescing are performed at this point.

**CFLOW optimize (second pass)** The same as the first pass, this phase iterates over the newly transformed code.

**Extended block optimizer** Peephole type optimizations are performed here, including constant propagation, redundant and dead expression elimination.

**Global instruction scheduling** Instructions are scheduled on a global level during this phase. The resultant schedule imposes global restrictions on the code.

**Local instruction scheduling** This pass schedules instructions locally, within the limitations of the schedule decided on by global scheduling.

**Localize global TNs** Global TNs that are able to become local variables are transformed at this stage. Local TNs are easier to handle during register allocation.[3]

**Global register allocation** This phase allocates registers on a global scale. Again, global dependences are introduced under which further allocations must conform.

**Local register allocation** Here, registers are allocated at the local level.

**Extended block optimizer post-process** Extended block optimizations are again performed after register allocation is complete.

**Local instruction scheduling** Further scheduling occurs during this phase, to allow the best schedule possible after all transformations have finished, for the actual generation of code.

**Generate code** Assembly language code is emitted for the target architecture.

## 6.2 Modified Code Generator

The work of this thesis has introduced SSA in the code generator of the ORC. Based on code originally written by Arthur Stoutchinin [35] for an STMicroelectronics architecture [34], the code was re-targeted for the Itanium processor by Stoutchinin, with later assistance from the author of this thesis. The SSA form is built after the global live range analysis phase, using Cytron *et al.*'s method from Section 4.1.1. There are six locations where the SSA can be removed, described in Table 6.1. Currently, methods 1 (Section 4.4.1) and 2 (Section 4.4.2) of Sreedhar *et al.* have been implemented. The choice of removal location can be made with a compile-time flag. The modified code generator's phases can be seen in Figure 6.3.

[3]A TN is a temporary name representing a variable instance in a program.

| Translation level | Where the SSA removal is performed |
|---|---|
| 1 | after extended block optimizer preprocessing |
| 2 | after first pass of control flow optimization |
| 3 | after if-conversion |
| 4 | after second pass of control flow optimization |
| 5 | after extended block optimization |
| 6 | after global scheduling |

Table 6.1: Description of SSA translation levels

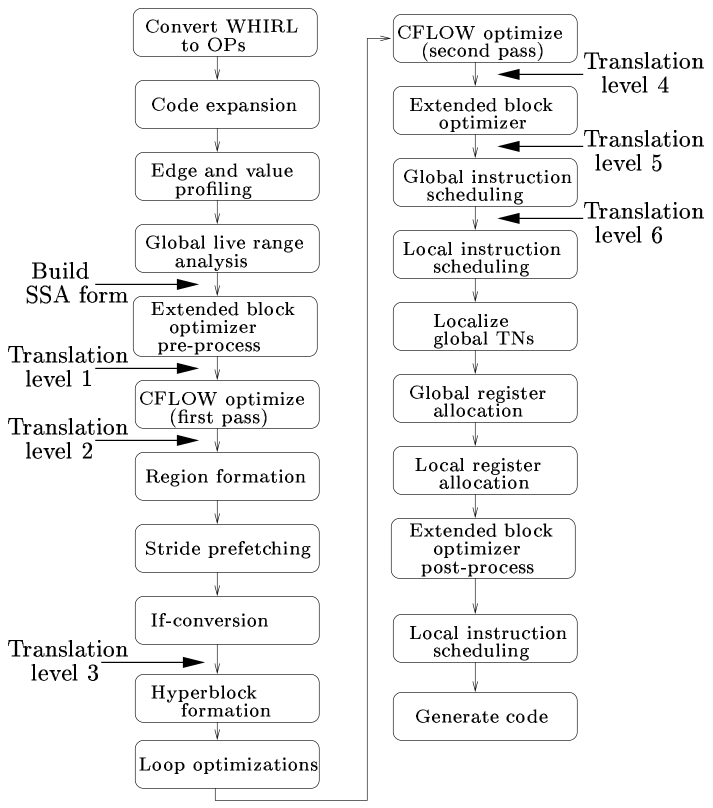


Figure 6.3: Modified ORC code generator

Consider again Figure 4.1, from Chapter 4. The sample code is presented here at various phases of the code generator using the modified compilation process at optimization level 2. Figures 6.4 and 6.5 graphically represent the CFG of the example code just before and immediately following SSA construction. Note that variables of the form $t_i$ are intermediate values that the compiler introduces, and GTNs are intermediate values relating to procedure preparation. In Figure 6.4, and for further figures, one can see an introduction of predicates well before the if-conversion phase of the compiler has occurred. Limited predication is used in earlier stages to handle some basic control flow issues.

Figure 6.6 pictures the CFG of the sample code after the SSA form has been removed following the extended block optimizer preprocessing phase. Figure 6.7 gives the sample code's representation after SSA removal subsequent to the first control flow optimization pass. Notice that the main difference between Figures 6.6 and 6.7 is the removal of node $B_7$ from Figure 6.7 and the inclusion of $B_7$'s code in node $B_6$.

Figure 6.8 indicates the form of the sample code following SSA removal after if-conversion. It is obvious that the code has taken on a very different form, with the union of many smaller nodes into several larger nodes. This process was facilitated by removing most of the control flow issues via if-conversion.

It turns out that, for this small example, the code representations after levels 4 and 5 of SSA removal do not change the CFG of level 3. Thus, Figure 6.8 is sufficient to show all three translation levels.

Figure 6.9 shows the code form following SSA removal after global scheduling has passed. This code is quite different from the code of earlier stages. Based on Intel's Itanium architecture, code scheduling is performed using code blocks of three, called *bundles*. If the scheduler cannot decide on three appropriate instructions to place within a bundle, *nop*'s are inserted. These operations that perform no task are undesirable, but often necessary, depending on the code's form at the time of scheduling.

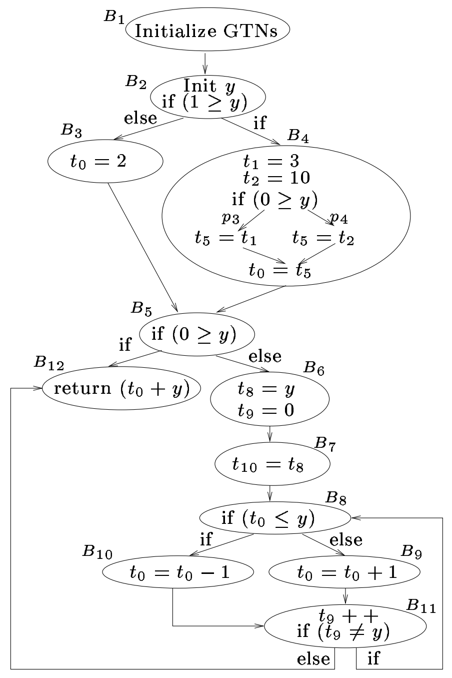


Figure 6.4: CFG for sample code before SSA construction

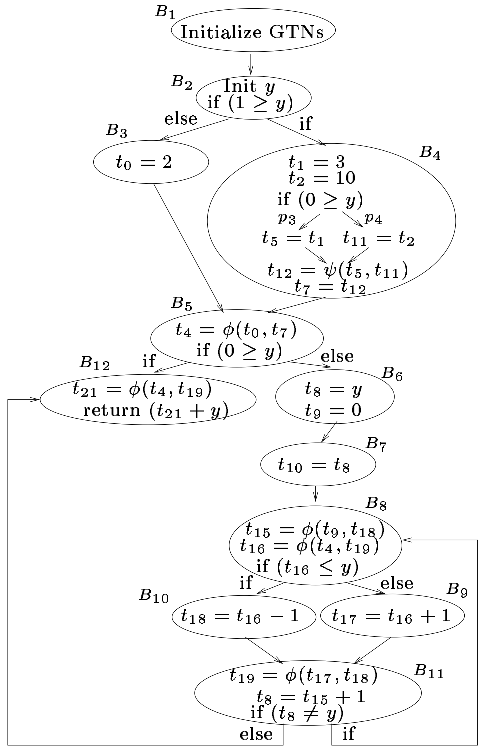


Figure 6.5: CFG for sample code after SSA construction

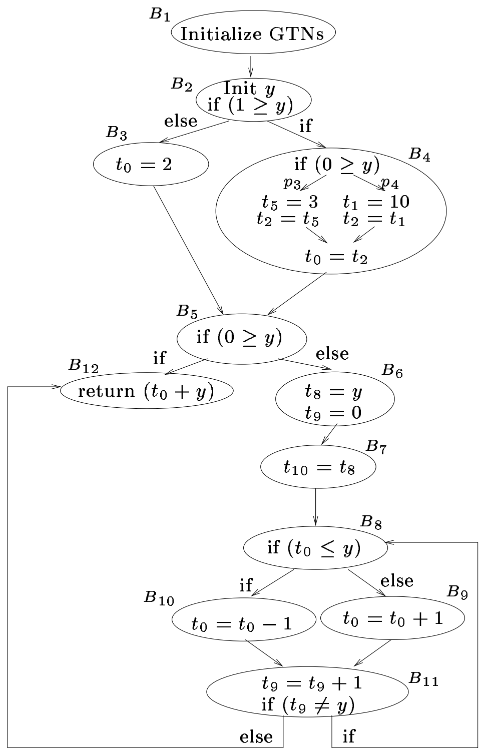


Figure 6.6: CFG for sample code after level 1 SSA removal

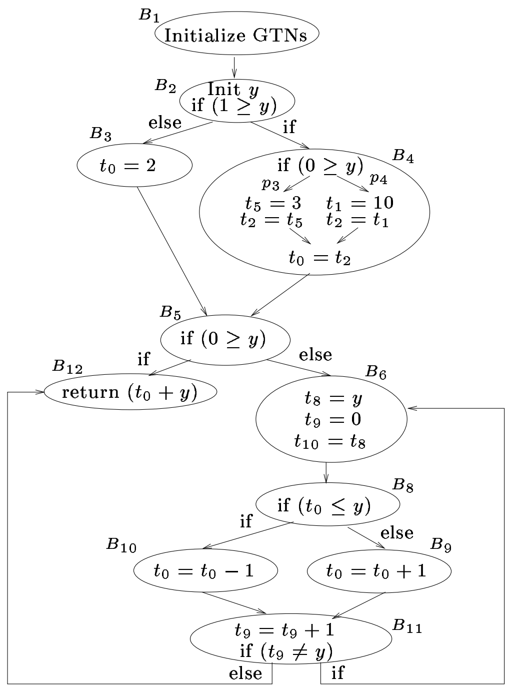


Figure 6.7: CFG for sample code after level 2 SSA removal

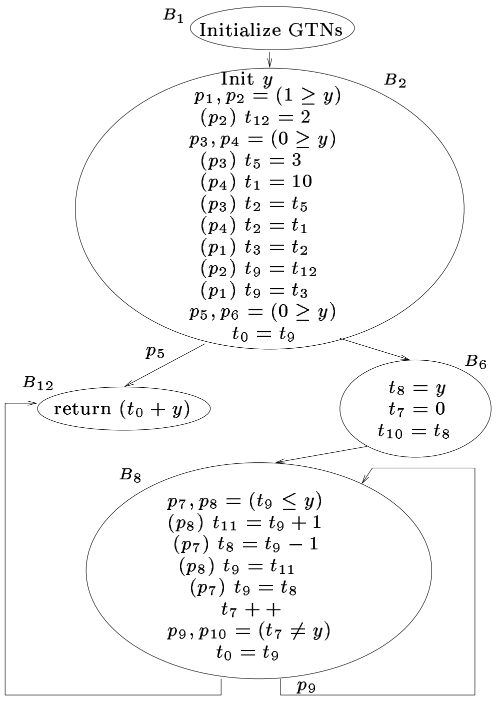


Figure 6.8: CFG for sample code after levels 3, 4 and 5 SSA removal

```
$B_1$:
($p_0$)    Initialize GTNs
($p_0$)    br: $B_2$
$B_2$:
($p_0$)    Init $y$
($p_0$)  $p_1, p_2 = (1 \geq y)$
($p_0$)    $nop$
($p_0$)  $p_3, p_4 = (0 \geq y)$
($p_2$)    $t_{12} = 2$
($p_0$)    $nop$
($p_0$)  $p_5, p_6 = (0 \geq y)$
($p_4$)    $t_1 = 10$
($p_2$)    $t_9 = t_{12}$
($p_3$)    $t_5 = 3$
($p_4$)    $t_2 = t_1$
($p_0$)    $nop$
($p_3$)    $t_2 = t_5$
($p_0$)    $nop$
($p_0$)    $nop$
($p_1$)    $t_3 = t_2$
($p_1$)    $t_9 = t_3$
($p_0$)    $nop$
($p_0$)    $t_0 = t_9$
($p_0$)    $nop$
($p_5$)    br: $B_{13}$
```

```
$B_6$:
($p_0$)    $t_7 = 0$
($p_0$)    $t_8 = y$
($p_0$)    $nop$
($p_0$)    $t_{10} = t_8$
($p_0$)    $nop$
($p_0$)    $nop$
$B_8$:
($p_0$)  $p_7, p_8 = (t_9 \leq y)$
($p_0$)    $t_7{+}{+}$
($p_0$)    $nop$
($p_7$)    $t_{14} = t_9 - 1$
($p_8$)    $t_{13} = t_9 + 1$
($p_0$)  $p_9, p_{10} = (t_7 \neq y)$
($p_0$)    $nop$
($p_7$)    $t_8 = t_{14}$
($p_8$)    $t_{11} = t_{13}$
($p_7$)    $t_9 = t_8$
($p_8$)    $t_9 = t_{11}$
($p_0$)    $nop$
($p_0$)    $t_0 = t_9$
($p_0$)    $nop$
($p_9$)    br: $B_8$
($p_0$)    br: $B_{12}$
$B_{13}$:
($p_0$)    br: $B_{12}$
$B_{12}$:
($p_0$)  return($t_0 + y$)
```

Figure 6.9: Sample code after level 6 SSA removal

Chapter 7

# Eliminating Redundant Join Set Computations in SSA

In this chapter, Cytron *et al.*'s $\phi$-placement algorithm from Section 4.1.1 is revisited. The computation of join sets is at the center of that SSA construction technique. Two factors make the join set computation an interesting component for analysis. First, Cytron *et al.*'s algorithm is still the most prevalent in today's production compilers. Additionally, the bulk of the work performed by this method involves the join set. These considerations make the join set computation appealing as a point for optimization.

Theorem 7 presents the fundamental principle of this discussion. However, some preliminary results are necessary.

**Theorem 5.** *Let $X$ be a subset of nodes in a $CFG = G(V, E)$ such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. Then,*

$$DF(X_1) \cup DF(X_2) = DF(X_1 \cup X_2) \tag{7.1}$$

*Proof.* Let $X$ be a subset of nodes in a $CFG = G(V, E)$ such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. By Equation 3.1, we know that:

$$DF(X) = \bigcup_{B_i \in X} DF(B_i)$$

Thus,

$$DF(X_1) \cup DF(X_2) = \bigcup_{B_i \in X_1} DF(B_i) \cup \bigcup_{B_i \in X_2} DF(B_i) = \bigcup_{B_i \in (X_1 \cup X_2)} DF(B_i)$$

$$= DF(X_1 \cup X_2)$$

□

**Theorem 6.** *Let $X$ be a subset of nodes in a $CFG = G(V, E)$ such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. Then,*

$$DF_i(X_1) \cup DF_i(X_2) = DF_i(X_1 \cup X_2) \tag{7.2}$$

*such that $DF_i$ is an element of the sequence that defines $DF^+$.*

*Proof.* Let $X$ be a subset of nodes in a $CFG = G(V, E)$ such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. The proof is by induction.
Base case: If $i = 1$, from Definition 19 we know that $DF_1 = DF(X)$. Hence:

$$DF_1(X_1) \cup DF_1(X_2) = DF(X_1) \cup DF(X_2)$$

From Theorem 5, we have:

$$DF(X_1) \cup DF(X_2) = DF(X_1 \cup X_2)$$

Thus,

$$DF_1(X_1) \cup DF_1(X_2) = DF(X_1 \cup X_2) = DF_1(X_1 \cup X_2)$$

Inductive case: Assume that $DF_i(X_1) \cup DF_i(X_2) = DF_i(X_1 \cup X_2)$ for $i = k$. Let $i = k + 1$. From Definition 19 we know that $DF_{i+1}(X) = DF(X \cup DF_i(X))$. Then,

$$\begin{aligned}
DF_{k+1}(X_1) \cup DF_{k+1}(X_2) &= DF(X_1 \cup DF_k(X_1)) \cup DF(X_2 \cup DF_k(X_2)) \\
&= DF(X_1) \cup DF(DF_k(X_1)) \cup DF(X_2) \cup DF(DF_k(X_2)) \\
&= DF(X_1 \cup X_2) \cup DF(DF_k(X_1 \cup X_2)) \\
&= DF(X_1 \cup X_2 \cup DF_k(X_1 \cup X_2) \\
&= DF_{k+1}(X_1 \cup X_2)
\end{aligned}$$

□

**Theorem 7.** *Let $X$ be a subset of nodes in a $CFG = G(V, E)$ such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. Then,*

$$J^+(X) = J^+(X_1) \cup J^+(X_2) \tag{7.3}$$

*Proof.* Let $X$ be a subset of nodes in a $CFG = G(V, E)$ such that $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$. Since $G$ is a finite graph, $DF^+(X)$ must be finite. We know from Definition 19 that $DF^+(X)$ is the limit of a sequence of elements $DF_i(X)$. Equivalently,

$$DF^+(X) = \lim_{i \to c} DF_i(X)$$

where $c$ is a constant. Recall from Theorem 1 that $DF^+(X) = J^+(X)$. Now

$$\begin{aligned} J^+(X_1) \cup J^+(X_2) &= DF^+(X_1) \cup DF^+(X_2) \\ &= \lim_{i \to c} DF_i(X_1) \cup \lim_{i \to c} DF_i(X_2) \\ &= \lim_{i \to c} [DF_i(X_1) \cup DF_i(X_2)] \\ &= \lim_{i \to c} DF_i(X_1 \cup X_2) \\ &= \lim_{i \to c} DF_i(X) \\ &= DF^+(X) \\ &= J^+(X) \end{aligned}$$

□

Consider two variables, $x$ and $y$, in a program. If $A(x) = A(y)$, then $J^+(A(x)) = J^+(A(y))$. Similarly, if $A(y) \subset A(x)$, then $J^+(A(y)) \subset J^+(A(x))$. Recall that $J^+(A(x)) = S_\phi(x)$, the minimum set of nodes where $\phi$-functions are required when constructing the SSA form of a program. Thus, if it can be shown that two variables have the same set of assignment nodes, then only one join set computation needs to be performed. Conversely, if one set of assignment nodes is a subset of another, two join set computations are still required. The intersection of the two sets (*i.e.*, the smaller set) will be calculated. However, the second computation (*i.e.*, the remainder of the larger set) will be smaller than the original.

The majority of the time in Cytron *et al.*'s worklist algorithm is spent iterating over the worklist $W$, and every variable has a worklist associated with it. In particular, the worklist has to be initialized for every variable $v$ to $A(v)$. Then, each element $B_i \in W$ is removed from $W$, and a $\phi$-function is inserted in every $B_j \in DF(B_i)$. As well, each $B_j$ that now contains a $\phi$-function is also inserted in $W$, and the process continues.

An initial analysis of individual SPEC CINT2000 benchmarks [17] indicates the opportunities for join set optimization. Table 7.1 gives a comparison between the number of times $A(y) \subseteq A(x)$ for two variables $x$ and $y$ and the number of worklists processed in the original implementation of the code generator's SSA construction. The percentage of work saved is the maximum number of entire worklists whose computation can be avoided by the elimination of redundant join set computations. All the calculations in this chapter were performed at SSA translation level 1 (refer to Table 6.1) and optimization level O2 for baseline results, using Sreedhar *et al.*'s translation method 2 and the individual benchmarks' *test* data set.

| Benchmark | Opportunities for optimization | Total number of worklists processed | Percentage of work saved |
|---|---|---|---|
| 164.gzip | 220 | 2787 | 7.89 |
| 181.mcf | 51 | 246 | 20.73 |
| 197.parser | 698 | 17567 | 3.97 |
| 254.gap | 3555 | 115686 | 3.07 |
| 255.vortex | 1294 | 37372 | 3.46 |
| 256.bzip2 | 348 | 12089 | 2.88 |
| Average | 1028 | 30958 | 3.32 |

Table 7.1: Opportunities for eliminating redundant join set computations in SPEC CINT2000 benchmarks

Let $x$ and $y$ be two variables such that $A(x) = A(y)$. If this condition is detected, the worklist algorithm only needs to iterate for one of the two variables, $W = A(x) = A(y)$. The major change comes upon insertion of a $\phi$-function for the variable being processed, since now there are two variables. Hence, for every $B_i \in W$ and $B_j \in DF(B_i)$, two $\phi$-functions are added in $B_j$, one each for $x$ and $y$. Performing the $\phi$-function insertion in this manner eliminates an entire worklist iteration, and thus an entire join set computation, along with cutting down on accesses to the dominance frontier data structure. In practice, $\phi$-functions are placed via a function call in the modified ORC SSA implementation in the code generator. Therefore, further savings can be achieved by removing a function call.

Table 7.2 shows the original opportunities presented in Table 7.1 where $A(x) = A(y)$, which is the best case scenario. In fact, more than half of the optimization possibilities explored can eliminate an entire worklist. The actual percentage of worklists avoided can be seen in Table 7.2.

Now let $x$ and $y$ be two variables such that $A(y) \subset A(x)$. One approach for handling this case is to split $A(x)$ into two smaller sets, $A(x_1) = A(y)$ and $A(x_2) = A(x) - A(y)$. This set division is possible since Theorem 7 determined that $J^+(A(x)) = J^+(A(x_1)) \cup J^+(A(x_2))$. Processing for $A(x_1)$ can be performed as normal. Consider here $A(x_2)$. The list of nodes that require a $\phi$-function because of $A(x_2)$ will have to be computed separately as a worklist for $x$. Thus, savings still exists, since $A(x_1)$ and $A(y)$ are combined, and the worklist for $A(x_2)$ requires fewer iterations since $A(x_2) < A(x)$.

| Benchmark | Opportunities for optimization | Total number of worklists processed | Percentage of worklists avoided |
|---|---|---|---|
| 164.gzip | 89 | 2787 | 3.19 |
| 181.mcf | 22 | 246 | 8.94 |
| 197.parser | 526 | 17567 | 2.99 |
| 254.gap | 1928 | 115686 | 1.67 |
| 255.vortex | 751 | 37372 | 2.01 |
| 256.bzip2 | 177 | 12089 | 1.46 |
| Average | 582.2 | 30958 | 1.88 |

Table 7.2: Instances where two join sets are equivalent in SPEC CINT2000 benchmarks

Consider the example in Figure 7.1. $A(x) = \{B_1, B_2, B_3, B_4, B_5\}$ and $A(y) = \{B_1, B_2, B_5\}$. Clearly, $A(y) \subset A(x)$. Now $A(x_1) = A(y) = \{B_1, B_2, B_5\}$ and $A(x_2) = A(x) - A(y) = \{B_3, B_4\}$. During the worklist algorithm, when $A(x)$ and $A(y)$ are compared, we can see that processing $A(y)$ will make up for most of the calculations also required by $A(x)$. Hence, when $\phi$-functions are placed for $y$ as part of the iteration for $A(y)$, we will know to also place $\phi$-functions for $x$. Remaining now is $A(x_2)$, the elements left over from the larger set $A(x)$. The worklist will have to iterate for $A(x_2)$, however this remaining set is smaller than the original $A(x)$. We will thus perform fewer computations.

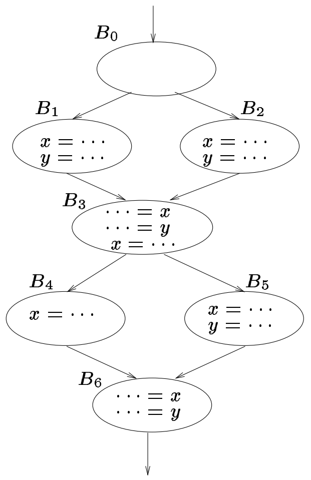


Figure 7.1: Example of sets of assignments for two variables, $x$ and $y$

Table 7.3 gives the number of instances from Table 7.1 that were actually $A(y) \subset A(x)$. This case is less attractive than the $A(x) = A(y)$ situation since it requires more work. However, if it can be shown that, generally, $A(x_2)$ is much smaller than $A(x)$, then the benefits could be significant. As can be seen by Table 7.3, we are on average saving 45% of the calculations by combining the portion of the two sets that intersect. This means that $A(x) - A(y)$ is generally 45% smaller than $A(x)$.

| Benchmark | Opportunities for optimization | Average difference in size of subsets | Average percentage saved |
|---|---|---|---|
| 164.gzip | 131 | 6.79 | 38.38 |
| 181.mcf | 29 | 3.45 | 42.99 |
| 197.parser | 172 | 2.65 | 52.10 |
| 254.gap | 1627 | 4.34 | 45.35 |
| 255.vortex | 543 | 5.00 | 53.14 |
| 256.bzip2 | 171 | 5.65 | 41.33 |
| Average | 445.5 | 5.58 | 45.55 |

Table 7.3: Instances where one join set is a subset of another join set in SPEC CINT2000 benchmarks

Checking the relationships between these sets requires some extra calculation, but much of the work is facilitated through existing data structures in the SSA code. The one-time expense incurred to build correspondences between individual sets should be worth the benefits achieved through minimizing join set computations.

This chapter evaluated opportunities for eliminating redundant join set computations in Cytron *et al.*'s $\phi$-placement algorithm. We feel that there is enough evidence to warrant implementing the join set optimization in the ORC, however time constraints leave this implementation outside of the scope of this thesis. Since the proof of concept tests were performed on a selection of the SPEC CINT2000 benchmark suite, this small optimization could be beneficial in other compilers that utilize Cytron *et al.*'s technique.

# Chapter 8

# Experimental Results

The SSA framework presented in Section 6.2 allows for optimizations in the code generator to take advantage of the benefits provided by the SSA form. However, if the gains allowed by SSA are outweighed by the cost of constructing and removing the SSA form, it may not be desirable to have SSA in the code generator. Since the work of this thesis does not introduce any further optimizations throughout the later phase SSA, the real assets of having SSA in the code generator are not realized. Therefore, if the SSA construction presented here is too expensive, future code transformations within SSA may be avoided. We will show that building the SSA form does not seriously degrade performance, thus making it a viable infrastructure upon which to introduce additional code transformations

This chapter presents the preliminary experimental results of including SSA in the code generator of the ORC. The experiments were run on an Itanium machine (HP Itanium2-2048 processor, 1GB memory). The ORC2.0 cross compiler used code compiled on an IA-32 machine (PentiumIII, 700MHz-128). Using the cross environment, four of the SPEC CINT2000 benchmarks did not behave as expected using the baseline compiler.[1] These discrepancies account for the omission of the problematic benchmarks from the results presented here.

For these experiments, $\phi$-functions were inserted using the insertion algorithm of Section 4.1.1. There are two SSA removal techniques (Sreedhar *et al.*'s methods 1 of Section 4.4.1 and method 2 of Section 4.4.2) that we could use for testing. Our experiments focus on method 2, which as discussed in this thesis is the superior algorithm. We present some comparisons with method 1 to justify this decision.

The results are broken down into 3 categories: compile-time and run-time results;

[1]175.*vpr* produced unexpected output; 186.*crafty* had trouble involving the linker; 252.*eon* could not find files that it needed to include; and the source code for 253.*perlbmk* contained a syntax error.

number of inserted instructions during SSA construction and removal; and number of actual instructions executed. All results were accumulated at optimization level O2 and used the individual benchmarks' *test* data set.

## 8.1 Timing Results

First of all, we would like to ensure that constructing the SSA form in the code generator does not unreasonably increase compile-time or degrade run-time performance. All timing results are presented in seconds, and are an average over 5 runs. The experiments were compiled while in single-user mode on the IA32 machine, and run while in single-user mode on the Itanium machine. The execution times were calculated using the UNIX *time* command.

The numbers listed in Table 8.1 compare SSA translation level 1 (refer to Table 6.1) and the baseline results.[2] On average, to compile a benchmark with later phase SSA included is 2.91% slower than without SSA. This difference is minor for the amount of extra work included. The execution time does not on average change with SSA included. Therefore, performance results have not been negatively affected by later phase SSA.

| Benchmark | Compile-time (baseline) | Compile-time (SSA) | Run-time (baseline) | Run-time (SSA) |
|---|---|---|---|---|
| 164.gzip | 22.29 | 22.24 | 1.69 | 1.71 |
| 181.mcf | 11.15 | 11.32 | 0.27 | 0.29 |
| 197.parser | 53.50 | 54.91 | 3.89 | 3.83 |
| 254.gap | 236.38 | 244.57 | 1.57 | 1.62 |
| 255.vortex | 144.74 | 148.69 | 5.69 | 5.70 |
| 256.bzip2 | 11.91 | 12.24 | 7.19 | 7.08 |
| Average | 80.00 | 82.33 | 3.38 | 3.37 |

Table 8.1: SPEC CINT2000 benchmark compile and run times (in seconds)

Tables 8.2, 8.3, 8.4 and 8.5 expand on the results of Table 8.1 for 164.*gzip*, 181.*mcf*, 254.*gap*, and 256.*bzip2* at the remaining translation levels, as well as showing the percentage differences in compile and run times.[3] Notice in Tables 8.2 and 8.3 that there is little deviation between the compile-times for the individual translation levels. These numbers indicate that the same cost ensues even when more work

[2]The omission of 176.*gcc* and 300.*twolf* from this, and subsequent tables, indicates that these benchmarks exhibited bugs at translation level 1 and higher.

[3]164.*gzip* and 254.*gap* had problems at translation level 6, and were thus omitted from Tables 8.2 and 8.4, respectively.

is performed, as the SSA form is maintained incrementally. Tables 8.4 and 8.5 show the expected trend, where the compile-time increases as more work is performed along later translation levels. Execution times vary greatly among the benchmarks. Tables 8.2, 8.4 and 8.5 present how the various translation levels consistently achieve better execution rates. However, Table 8.3 shows that *mcf* performs poorly across translation levels. These results are still preliminary, as the full benefits are not expected to be realized until after optimizations that exploit the properties of SSA are implemented. At this point, since we want to maintain the SSA form as long as possible, we just need to ensure that the individual translation levels do not overwhelmingly degrade performance.

| Translation level | Compile-time | Time increase (%) | Run-time | Speedup (%) |
|---|---|---|---|---|
| 0 | 22.29 | - | 1.69 | - |
| 1 | 22.24 | -0.22 | 1.71 | -1.18 |
| 2 | 22.50 | 0.94 | 1.65 | 2.37 |
| 3 | 22.95 | 2.96 | 1.65 | 2.37 |
| 4 | 23.23 | 4.22 | 1.65 | 2.37 |
| 5 | 23.10 | 3.63 | 1.67 | 1.18 |

Table 8.2: Compile and run times for *gzip* (in seconds)

| Translation level | Compile-time | Time increase (%) | Run-time | Speedup (%) |
|---|---|---|---|---|
| 0 | 11.15 | - | 0.27 | - |
| 1 | 11.32 | 1.53 | 0.29 | -7.41 |
| 2 | 11.29 | 1.26 | 0.27 | 0 |
| 3 | 11.18 | 0.27 | 0.28 | -3.70 |
| 4 | 11.37 | 1.97 | 0.27 | 0 |
| 5 | 11.28 | 1.17 | 0.28 | -3.70 |
| 6 | 11.45 | 2.69 | 0.27 | 0 |

Table 8.3: Compile and run times for *mcf* (in seconds)

| Translation level | Compile-time | Time increase (%) | Run-time | Speedup (%) |
|---|---|---|---|---|
| 0 | 236.38 | - | 1.57 | - |
| 1 | 244.57 | 3.46 | 1.62 | -3.18 |
| 2 | 245.99 | 4.07 | 1.30 | 17.20 |
| 3 | 249.03 | 5.35 | 1.92 | -22.29 |
| 4 | 254.08 | 7.49 | 1.41 | 10.19 |
| 5 | 256.68 | 8.59 | 1.32 | 15.92 |

Table 8.4: Compile and run times for *gap* (in seconds)

| Translation level | Compile-time | Time increase (%) | Run-time | Speedup (%) |
|---|---|---|---|---|
| 0 | 11.91 | - | 7.19 | - |
| 1 | 12.24 | 2.77 | 7.08 | 1.53 |
| 2 | 12.68 | 6.47 | 7.11 | 1.11 |
| 3 | 12.83 | 7.72 | 7.13 | 0.83 |
| 4 | 13.01 | 9.24 | 7.10 | 1.25 |
| 5 | 12.91 | 8.40 | 6.17 | 14.19 |
| 6 | 13.44 | 12.85 | 6.18 | 14.05 |

Table 8.5: Compile and run times for *bzip2* (in seconds)

## 8.2 Inserted Instructions

The next interesting measure of how the later phase SSA performed is the number of inserted instructions. Throughout the SSA algorithm, there are two distinct opportunities for additional instructions to be included in the intermediate code. Both $\phi$ and $\psi$ functions are inserted, and then copies are included to remove these unexecutable instructions. It is obviously desirable that the number of extra instructions not greatly hinder the baseline performance. The number of actual instructions executed is discussed in Section 8.3.

Table 8.6 gives a summary of the number of $\phi$, $\psi$, and copy instructions added through the process of building and removing the SSA form for translation level 1. As will be seen in Section 8.3, the weight of these inserted instructions is negligible.

| Benchmark | # of $\phi$ and $\psi$ functions inserted | # of copies inserted |
|---|---|---|
| 164.gzip | 1089 | 437 |
| 181.mcf | 230 | 67 |
| 197.parser | 2364 | 421 |
| 254.gap | 13682 | 6020 |
| 255.vortex | 6112 | 1058 |
| 256.bzip2 | 771 | 211 |

Table 8.6: Number of $\phi$, $\psi$ and copy instructions inserted in the SPEC CINT2000 benchmarks

Tables 8.7, 8.8, 8.10, and 8.11 expand on the results of Table 8.6 for the remaining translation levels. In is interesting to note that, on average, more copies are needed to remove the $\phi$ and $\psi$ functions as the translation levels increase. As the SSA form is maintained longer, more transformations can be performed. Therefore, the analysis that results in the insertion of copies can become more difficult, resulting in more copies being required to ensure correctness.

For comparative purposes, we have also included the results for Sreedhar *et al.*'s translation method 1 (Section 4.4.1) in Table 8.9.[4] Method 2 outperforms method 1 in the number of $\phi$-copies inserted for the individual translation methods, and thus is justified as the method of interest. Method 1 inserts many more copies than are necessary.

| Translation level | # of $\phi$-functions inserted | # of copies inserted | # of $\psi$-functions inserted | # of copies inserted |
|---|---|---|---|---|
| 1 | 989 | 269 | 100 | 168 |
| 2 | 989 | 269 | 100 | 168 |
| 3 | 989 | 304 | 100 | 232 |
| 4 | 989 | 309 | 100 | 232 |
| 5 | 989 | 309 | 100 | 228 |

Table 8.7: Number of $\phi$, $\psi$ and copy instructions inserted at individual translation levels in *gzip*

| Translation level | # of $\phi$-functions inserted | # of copies inserted | # of $\psi$-functions inserted | # of copies inserted |
|---|---|---|---|---|
| 1 | 193 | 10 | 37 | 57 |
| 2 | 193 | 32 | 37 | 57 |
| 3 | 193 | 27 | 37 | 87 |
| 4 | 193 | 25 | 37 | 87 |
| 5 | 193 | 25 | 37 | 75 |
| 6 | 222 | 25 | 37 | 75 |

Table 8.8: Number of $\phi$, $\psi$ and copy instructions inserted at individual translation levels in *mcf*

| Translation level | # of $\phi$-functions inserted | # of copies inserted | # of $\psi$-functions inserted | # of copies inserted |
|---|---|---|---|---|
| 1 | 193 | 619 | 37 | 57 |
| 2 | 193 | 624 | 37 | 57 |
| 3 | 193 | 564 | 37 | 87 |
| 4 | 193 | 530 | 37 | 87 |
| 5 | 193 | 530 | 37 | 75 |

Table 8.9: Number of $\phi$, $\psi$ and copy instructions inserted at individual translation levels in *mcf* for Sreedhar's *et al.*'s translation method 1

[4]Translation level 6 had a problem with the linker, and is thus excluded from these results.

| Translation level | # of $\phi$-functions inserted | # of copies inserted | # of $\psi$-functions inserted | # of copies inserted |
|---|---|---|---|---|
| 1 | 10909 | 1959 | 2773 | 3830 |
| 2 | 10909 | 2172 | 2773 | 3848 |
| 3 | 10909 | 2292 | 2773 | 4787 |
| 4 | 10909 | 2165 | 2773 | 4791 |
| 5 | 10909 | 2209 | 2773 | 4762 |

Table 8.10: Number of $\phi$, $\psi$ and copy instructions inserted at individual translation levels in *gap*

| Translation level | # of $\phi$-functions inserted | # of copies inserted | # of $\psi$-functions inserted | # of copies inserted |
|---|---|---|---|---|
| 1 | 673 | 86 | 98 | 125 |
| 2 | 673 | 87 | 98 | 125 |
| 3 | 673 | 114 | 98 | 175 |
| 4 | 673 | 111 | 98 | 175 |
| 5 | 673 | 113 | 98 | 171 |
| 6 | 818 | 113 | 98 | 171 |

Table 8.11: Number of $\phi$, $\psi$ and copy instructions inserted at individual translation levels in *bzip2*

## 8.3 Executed Instructions

The final category of measurements is the actual number of instructions executed, or retired, at runtime. These figures were obtained using the hardware performance monitoring tool *pfmon* [13]. We want to measure the increased number of instructions executed using SSA, hoping the additions do not overwhelm the original code generator's results.

Table 8.12 indicates the differences between number of executed instructions at the baseline level versus those executed at translation level 1. A percentage indicating number of extra instructions is also included. In general, there are only an extra 0.51% of instructions executed when later phase SSA is included

Tables 8.13, 8.14, 8.16, and 8.17 expand on the results of Table 8.12 for the remaining translation levels. We can see from these tables that there is no significant change in the number of retired instructions as we proceed through the translation levels.

Comparatively, Table 8.15 gives the number of retired instructions for *mcf* using Sreedhar *et al.*'s translation method 1. On average, the naive translation method executes an extra 11% of instructions over the algorithm that uses interference graph

| Benchmark | Baseline | SSA | % of increase in instructions over baseline |
|---|---|---|---|
| 164.gzip | 4,226,267,378 | 4,444,772,461 | 5.17 |
| 181.mcf | 292,144,926 | 296,331,818 | 1.43 |
| 197.parser | 5,500,841,729 | 5,513,781,510 | 0.24 |
| 254.gap | 1,624,134,732 | 1,627,696,252 | 0.22 |
| 255.vortex | 13,186,363,361 | 13,199,873,707 | 0.10 |
| 256.bzip2 | 14,270,261,637 | 14,215,398,582 | -0.38 |
| Average | 6,516,668,961 | 6,549,642,388 | 0.51 |

Table 8.12: Number of executed instructions for the SPEC CINT2000 benchmarks

updates.

The results presented in this chapter have shown that the later phase SSA framework adds a minimal amount of compile-time and few additional executed instructions for the SPEC CINT2000 benchmarks. As well, the run-time is not compromised by the SSA inclusion. Therefore, working with the later phase SSA by introducing further code optimizations is a competitive option for the ORC.

| Translation level | # of retired instructions | % increase over baseline |
|---|---|---|
| Baseline | 4,226,267,378 | - |
| 1 | 4,444,772,461 | 5.17 |
| 2 | 4,475,198,347 | 5.89 |
| 3 | 4,489,361,441 | 6.23 |
| 4 | 4,492,050,665 | 6.29 |
| 5 | 4,516,763,080 | 6.87 |

Table 8.13: Number of executed instructions at individual translation levels in *gzip*

| Translation level | # of retired instructions | % increase over baseline |
|---|---|---|
| Baseline | 292,144,926 | - |
| 1 | 296,331,818 | 1.43 |
| 2 | 296,331,791 | 1.43 |
| 3 | 296,002,885 | 1.32 |
| 4 | 295,647,938 | 1.20 |
| 5 | 293,062,433 | 0.31 |
| 6 | 292,946,658 | 0.27 |

Table 8.14: Number of executed instructions at individual translation levels in *mcf*

| Translation level | # of retired instructions | % increase over baseline |
|---|---|---|
| Baseline | 292,144,925 | - |
| 1 | 331,373,448 | 13.43 |
| 2 | 329,715,993 | 12.74 |
| 3 | 328,431,952 | 12.42 |
| 4 | 327,940,049 | 12.25 |
| 5 | 324,111,065 | 10.94 |

Table 8.15: Number of executed instructions at individual translation levels in *mcf* using Sreedhar's *et al.*'s translation method 1

| Translation level | # of retired instructions | % increase over baseline |
|---|---|---|
| Baseline | 1,624,134,732 | - |
| 1 | 1,627,696,252 | 0.22 |
| 2 | 1,627,086,081 | 0.18 |
| 3 | 1,634,621,472 | 0.65 |
| 4 | 1,633,378,414 | 0.57 |
| 5 | 1,633,645,871 | 0.59 |

Table 8.16: Number of executed instructions at individual translation levels in *gap*

| Translation level | # of retired instructions | % increase over baseline |
|---|---|---|
| Baseline | 14,270,261,637 | - |
| 1 | 14,215,398,582 | -0.38 |
| 2 | 14,201,901,874 | -0.48 |
| 3 | 14,216,218,572 | -0.38 |
| 4 | 14,230,515,987 | -0.28 |
| 5 | 14,261,806,865 | -0.06 |
| 6 | 14,274,974,881 | 0.03 |

Table 8.17: Number of executed instructions at individual translation levels in *bzip2*

# Chapter 9

# Future Work

Since we expect some improvements in efficiency using Sreedhar *et al.*'s third translation technique (Section 4.4.3), it should be implemented in the ORC. Then, comparisons can be made with the two techniques already in place.

When the full framework is established, it would be of great interest to add additional optimizations into the code generator of the ORC. Besides the code transformations already in place, further benefits could be realized with the inclusion of specific optimizations geared towards the SSA form. As discussed in Chapter 8, the current implementation does not hinder performance, but does nothing to improve it. Optimizations that take full advantage of SSA could certainly result in experimental gains.

The enhancement to Cytron *et al.*'s $\phi$-placement algorithm presented in Chapter 7 gives another opportunity for future work. The evidence is strong that this optimization of the traditional algorithm will produce a decreased amount of iterations through the worklist algorithm. Thus, an improvement in compile-time is possible.

A long term goal of this project is to maintain the later phase SSA algorithm even further in the code generator, at least through local instruction scheduling. Eventually, a method may be discovered for handling SSA during register allocation.

Finally, it would be interesting to evaluate the effects of later phase SSA on other compilers. However, compilers intended for architectures that support predication should benefit from this algorithm more than others.

# Chapter 10

# Conclusion

This thesis has presented a comprehensive study into the unique properties and wide ranging capabilities of the Static Single Assignment form. Commonly used to ease dataflow analysis, SSA is a powerful representation that produces many benefits for the code optimizations supported by it. However, not all transformations have traditionally been able to avail of the SSA form. In particular, architectures that implement predication have avoided SSA in the later stages of the compiler after if-conversion removes conditional expressions. Unfortunately, such later phase code transformations have not benefited from the elegant framework provided by the SSA representation.

Throughout the course of this thesis, $\psi$-SSA, a mechanism for dealing with SSA at a later compiler phase, has been presented. $\psi$-SSA combines traditional SSA $\phi$-placement with new ideas for handling predicated execution to produce an entire SSA algorithm. Additionally, implementation details using the Open Research Compiler were disclosed. The framework for the code generator has been shown to not impose serious performance penalties on the baseline compiler, thus making the later phase SSA algorithm a viable starting point for further code optimizations.

As well, an improvement to a well-known $\phi$-placement technique was suggested. Throughout this method, worklists are used to decide where $\phi$-functions should be inserted. We have shown that entire worklists can be eliminated and the number of elements in some remaining worklists decreased using our augmentation. The opportunities for optimization were explored for the SPEC CINT2000 benchmark suite on the ORC. The validity of the enhancement allows future implementations to incorporate the small, but significant, change.

This thesis has demonstrated the usefulness of SSA at a later stage of the compilation process. Since most production compilers avoid SSA in the code generator, we believe that many opportunities for improving the quality of code produced are lost. Using the $\psi$-SSA algorithm, a new framework for the code generator of the ORC is now available. We hope that this work is the beginning of a renewed focus on utilizing SSA to its full potential in later compiler phases.

# Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Calif., USA, 1986.

[2] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann Publishers, Calif., USA, 2001.

[4] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the Conference on Principles of Programming Languages*, pages 1–11, 1988.

[5] B. Bilardi and K. Pingali. The static single assignment form and its computation. Technical report, Cornell University, 1999.

[6] G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50(3):375–425, 2003.

[7] P. Briggs, K.D. Cooper, T.J. Harvey, and L.T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software-Practice and Experience*, 28(8):859–881, July 1998.

[8] Z. Budimlić, K.D. Cooper, T.J. Harvey, K. Kennedy, T.S. Oberg, and S.W. Reeves. Fast copy coalescing and live-range identification. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume 37 of *ACM SIGPLAN Notices*, pages 25–32, May 2002.

[9] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 245–255, Oct. 1999.

[10] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.

[11] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the Symposium on Compiler Construction*, volume 17 of *ACM SIGPLAN Notices*, pages 98–105, June 1982.

[12] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the Conference on Principles of Programming Languages*, pages 55–66, 1991.

[13] Hewlett-Packard Development Company. *pfmon.* http://www.hpl.hp.com/research/linux/perfmon/pfmon.php4, 2003.

[14] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual.* Volume 3: Instruction Set Reference, Revision 2.0, 2001.

[15] Intel Corporation. http://www.intel.com/, 2003.

[16] Intel Corporation. *The Intel Itanium Processor.* http://www.intel.com/products/server/processors/server/itanium/, 2003.

[17] The Standard Performance Evaluation Corporation. *The SPEC CINT2000 Benchmark Suite.* http://www.spec.org/cpu2000/, 2003.

[18] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Jan. 1989.

[19] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[20] R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume 28 of *ACM SIGPLAN Notices*, pages 36–45, June 1993.

[21] R.K. Cytron and J. Ferrante. Efficiently computing $\phi$-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, May 1995.

[22] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 185–194, 1985.

[23] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[24] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, 1992.

[25] S.S. Muchnick and N.D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall Inc., New Jersey, USA, 1981.

[26] Chinese Academy of Sciences. http://english.cas.ac.cn/english/, 2003.

[27] J.C.H. Park and M. Schlankser. On predicated execution. Technical report, Hewlett-Packard Software and Systems Laboratory, 1991.

[28] K. Pingali and G. Bilardi. APT: A data structure for optimal control dependence computation. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, pages 32–46, June 1995.

[29] B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle. The Cydra-5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *Computer*, 22(1):12–35, Jan. 1989.

[30] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the Conference on Principles of Programming Languages*, pages 12–27, 1988.

[31] SourceForge. *Open Research Compiler.* http://ipf-orc.sourceforge.net/, 2003.

[32] V.C. Sreedhar and G.R. Gao. A linear time algorithm for placing $\phi$-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, 1995.

[33] V.C. Sreedhar, R.D.-C. Ju, D.M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Static Analysis 1999*, volume 1694 of *Lecture Notes in Computer Science*, pages 194–210.

[34] STMicroelectronics. http://www.st.com/, 2003.

[35] A. Stoutchinin. *Code extension to the Open Research Compiler.* Unpublished, 2003.

[36] A. Stoutchinin and F. de Ferriere. Efficient static single assignment form for predication. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 172–181, Dec. 2001.

[37] Open64 Compiler Tools. *WHIRL Documentation.* http://open64.sourceforge.net/documentation.html, 2003.

[38] Trimaran. *An Infrastructure for Research in Instruction-Level Parallelism.* http://www.trimaran.org/, 2003.

[39] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, Calif., USA, 1986.