Delft University of Technology
Master's Thesis in Computer Engineering

SSA Back-Translation: Faster Results with Edge Splitting and Post Optimization

Maarten Faddegon

embedded software

TU Delft Delft University of Technology

SSA Back-Translation: Faster Results with Edge Splitting and Post Optimization

Master's Thesis in Computer Engineering

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Maarten Faddegon
me@maartenfaddegon.nl

May 31st, 2011

Author
Maarten Faddegon (me@maartenfaddegon.nl)
Title
SSA Back-Translation: Faster Results with Edge Splitting and Post Optimization
MSc presentation
June 14th, 2011

Graduation Committee

prof. dr. K.G. Langendoen	Delft University of Technology
dr. M. Beemster	ACE Associated Compiler Experts
dr. G.N. Gaydadjiev	Delft University of Technology

Abstract

A compiler translates one representation of a software program into another. Beside translation compilers often have other tasks such as optimizating the result and warning the programmer for mistakes. Internally a compiler uses an Intermediate Representation (IR) for analysis and manipulation of the program at hand.

Data dependencies in most programming languages are implicit. Some compilers use an IR in Static Single Assignment (SSA) in which each local variable is only defined once to simplify analysis of data dependencies. If the number of assignments in the IR is not restricted, it is said to be in normal form. Input of a compiler is in normal form and translation is needed to bring the IR in SSA form.

SSA-form contains phi functions to merge values based on control flow. After optimizations on SSA-form are performed it is not trivial to translate SSA-form back to normal form because the properties of phi nodes cannot be translated directly to processor instructions. The algorithms of Briggs and Sreedhar are the two major methods of back-translation.

This thesis presents a modification that can be applied to the methods of Briggs and Sreedhar. The original methods append copy instructions to the end of existing source blocks. The presented modification splits edges between source and target by inserting phiblocks where the algorithms of Sreedhar and Briggs emit copy operations to replace phi functions.

For this study a bridge between LLVM and CoSy was built such that LLVM can be used for the optimizations on SSA-form and CoSy for the post back-translation optimizations. Four back-translation algorithms are implemented in CoSy. The methods are compared through experiments with six testcases from the SPEC benchmark suite.

On average the presented modification reduces the execution time of the resulting code with 5% for Briggs' method and 3% for Sreedhar's method. Experiments also show that the result of back-translation with or without phiblocks is suboptimal: repeating optimizations after back-translation that were already done on the IR in SSA-form can reduce the execution time on average with 18%.

Preface

This thesis is the final project for receiving the Master of Science degree in Computer Engineering programme at Delft University of Technology. It was carried out at ACE Associated Compiler Experts between September 2010 and May 2011.

During my stay at ACE I was asked to look into the mapping of LLVM's intermediate representation onto CoSy's intermediate representation. Soon I was facing the problem of back-translating static single assignment form. After reading a number of papers on the subject I implemented the two main algorithms for back-translation and started experimenting with variations on these algorithms. This led to an interesting modification that enables the generation of faster code.

For their support, advice and guidance and the opportunity to work with the CoSy technology I would like to thank Marcel, Toru and the other developers of ACE.

I would like to thank Koen for his advice and for serving as my supervisor at Delft University of Technology even during his sabbatical.

For their moral and financial support during my studies I like to thank my parents Hans and Ingrid.

Maarten Faddegon

Amsterdam, The Netherlands
May 31st, 2011

Contents

Chapter 1

Introduction

Compilers translates one representation of a software program into another. Beside translation compilers often have other tasks such as optimizing the result and warning the programmer for mistakes. Internally a compiler uses an Intermediate Representation (IR) for analysis and manipulation of the program.

LLVM is an open-source compiler popular amongst academic research projects [11, 10]. CoSy is a compiler-framework build by ACE. Both compilers are based on a modern design modelled around one intermediate representation (IR). However, the design of the intermediate representation of LLVM and CoSy differs.

During my stay at ACE a bridge between LLVM and CoSy was needed that maps the IR of LLVM onto CoSy's IR (CCMIR). One of the largest differences between the compilers is that the IR and the algorithms used by LLVM are based on Static Single Assignment (SSA) form. In the SSA representation each variable has only one definition, the rationale behind this limitation is that the compiler can analyse the IR faster and/or easier. Therefore, LLVM performs most optimizations on the SSA form.

The front end of the LLVM compiler translates the IR into SSA form. The IR of CoSy is not in SSA form (from here on not-SSA-form will be called normal form). Before the codegenerator in the back-end of the compiler can emit processor instructions, the SSA form is to be back-translated to normal form. Various methods for the back-translation of SSA form exist but the two main methods of back-translation are the methods of Briggs[3] and Sreedhar[13]. The research of this thesis concerns the optimality of these methods. The shared characteristic of these methods is that they glue together variables by appending copy instructions at the end of a branch.

Since optimization is performed before before back-translation, abundant use of copy instructions may have a negative impact on execution time. By repeating optimizations before and after back-translation this thesis answers the question:

- Can the back-translation methods of Briggs and Sreedhar undo optimizations that are performed before going out of SSA?

Furthermore, this thesis presents a modification that can be applied to the methods of Sreedhar and Briggs that speeds up results by reducing the number of copy instructions executed by placing them in separate blocks (named phiblocks). when fewer copy instructions need to be executed the resulting program is faster. This thesis provides answers to the following questions:

- What is the effect of placing instructions in phiblocks compared to the conventional methods of Sreedhar and Briggs on the execution time of the resulting code?
- Previous research by Sassa[12] showed that Sreedhar's method is superior to Briggs' method. Is this also true when the phiblock modification is applied?

To find answer to these questions both methods of back-translation and the variations on these methods are implemented. Six testcases from the SPEC suite are used to obtain empirical data. Furthermore a number of small cases are used as examples for the reasoning behind the insertion of phiblocks.

The remainder of this thesis is structured as follows: Background information of normal form, SSA and other terminology used in this document are explained in Chapter 2. Chapter 3 provides short summaries of previous work in the field of SSA back-translation. In Chapter 4 a novel modification is described to the existing methods. The goal of this modification is to improve the final result when the method is used in combination with optimization techniques after back-translating. How the different back-translation algorithms are compared is described in Chapter 5 The case studies are presented in Chapter 6. Chapter 7 contains an analysis of the back-translation methods from a software engineering viewpoint. The results of the tests from the SPEC benchmark suite are presented in Chapter 8. The last chapter describes conclusions and presents answers to the questions based on the results of the case studies and benchmarks.

Chapter 2

Background

This chapter provides background information on compilers, covering concepts such as normal form, static single assignment (SSA) form and other terminology.

2.1 Compiler

A compiler is a program that translates one representation of a software program into a different representation. For example, a C-compiler translates a piece of code written in the C programming language into processor specific assembly.

Beside direct translation, compilers often have other tasks such as optimizing the result for speed and/or size. Compilers can also analyze the code and warn the programmer for mistakes.

The input representation is targeted at the human programmer. The output representation is aimed at the processor architecture. Analysis and transformations on these representations is difficult because control flow and data dependencies are implicit. Therefore a third representation is used. This representation is only used internally by the compiler and is called intermediate representation (IR). Input, output and IR of a compiler share the same meaning. But their representation is different because they serve different goals.

Different compilers use different intermediate representations. The design of an IR has impact on the design of the algorithms that work on the IR. Therefore the design of the IR affects the speed of the compiler and the complexity of the design of a transformation on the IR.

In this document two intermediate representations are studied: The IR of the LLVM compiler and CoSy's IR (CCMIR). LLVMIR is represented in Static Single Assignment (SSA) form. CoSy works with an IR in normal form. The next two sections define intermediate representation in normal form and SSA from.

2.2 Normal Form

Before SSA-form can be described, it should be clear what intermediate representation in normal form is.

An important property of an intermediate representation is to enable the compiler to analyse the control flow. In 1970 Allen [1] described the use of basic blocks and a control flow graph, which is used today in almost all compilers. A short summary of the basic concepts is provided in this section.

A function is divided in linear sequences of statements. These sequences are called basic blocks. The first statement in the basic block is the only entry point. The last statement in the basic block is a terminator: a statement that controls which statement is executed next. Examples of terminators are goto and if-then-else statements.

A basic block has a number of possible predecessors and possible successors. A control flow graph (CFG) is a directed graph. The nodes in the CFG represent the basic blocks. The edges represent possible control flow between the basic blocks.

2.3 Static Single Assignment

SSA-form is a representation in which for each variable statically there may only be one assignment to that variable. The difference between static and dynamic assignment is explained in the next section. SSA-form is usually only applied on local variables. Global variables are excluded because they can be accessed from different fuctions and even from code not analysed by the compiler.

We take a look at a small example, in which we want to call a function and then add 4 to the result. Different representations are possible. One of the possible representations in normal form that fits the description is:

$$\begin{aligned} a &:= f() \\ a &:= a + 4 \end{aligned}$$

In this example variable a is assigned to twice. Multiple assignments to the same variable are not allowed in SSA-form. To translate this example to SSA-form, a second variable is needed.

$$\begin{aligned} a_1 &:= f() \\ a_2 &:= a_1 + 4 \end{aligned}$$

But, most programs are more complex than the example above. They can contain branches and loops. Section 2.4 and 2.5 explain how the dataflow of such constructs is represented in SSA-form.

2.4 Dynamic Assignment

An assignment may be inside a loop. For each iteration of the loop, an assignment to the same variable is done. An assignment during execution of the program is called a *dynamic* assignment. An assignment in the intermediate representation is called a *static* assignment. Thus, an assignment inside a loop can be assigned to statically once, while it is assigned to dynamically as many times as the loop iterates. Assigning multiple times to the same variable dynamically does not violate the *static* single assignment restriction.

Take a look at the following example. It illustrates the difference between static and dynamic assignment.

$$
\begin{aligned}
&b := 1 \\
&b := 2 \\
&\texttt{loop } 10 \texttt{ times} : \\
&\qquad a := b \\
&\texttt{f}(a, b)
\end{aligned}
$$

Variable a is statically assigned to once. Dynamically it is assigned to 10 times. (Even though it does not make much sense to assign the same value 10 times).

Assignments to the other variable are outside the loop. Therefore the number of static and dynamic assignments is equal. This example is not in SSA-form, because the number of static assignments to b is larger than 1.

In SSA-form, multiple uses (reading the content) of variables are allowed, static as well as dynamic.

2.5 Branches and Phi-functions

Almost any program used in the real world needs at a certain point to make a decision. Therefore it contains branches. This can be found in constructs such as conditionally executed code. Or whether to continue or to step out of a loop.

An often occurring possibility is that in these branches different assignments to the same variable are performed. For example, imagine two branches in which a value is assigned to variable x. For now we define these branches as the left and right branch. When the two branches merge, x is used. Figure 2.1 provides a graph of this case.

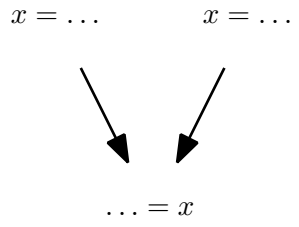


Figure 2.1: Example of two merging branches in normal form.

To translate to SSA, the assignments in the branches must be done to different variables, say to x_1 and x_2, respectively. But, which of these variables should be used after the branches? When the previous branch was the left branch, x_1 should be used. In the other case x_2 is to be used. To handle this variable merging, a function is used that is aware from which of the preceding blocks the controlflow comes. These special functions are called phi-functions, and are represented with the symbol 'ϕ'. Figure 2.2 shows the merging-branches example represented in SSA-form with the use of a phi-function.

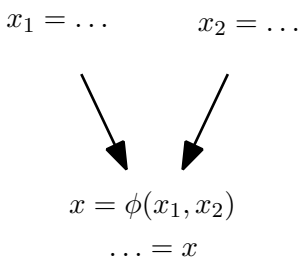


Figure 2.2: Example of two merging branches in SSA-form.

2.5.1 Special Properties of Phi Functions

In the example the phi function is the first (and only) statement of the block in which the control flow comes together and the variables are merged. Phi functions are always placed before any other statement in a block.

The phi functions of a basic block are an atomic, read-before-write action. This means that upon entering a basic block first the arguments of all phi functions are read. Then the target variables of all phi functions are written.

To show how this can be used we take a look at the notation of an example about swapping the values of two variables. We focus on a basic block that is the core of a loop. This block has two possible predecessors: flow can come from the block before the loop and also from the block itself. Before the

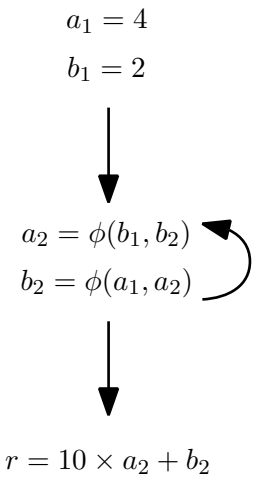


Figure 2.3: Swapping variables with phi functions.

loop variables a_1 and b_1 are initialized. In SSA-form it is possible to swap the values of a_2 and b_2 without the explicit use of a temporary variable, as illustrated by Figure 2.3.

An important property that is used in the swap example and the example of Figure 2.2 is that a phi function "knows" which basic block was executed previously.

The predecessor awareness, atomicity and read-before-write property of phi functions make that they cannot be translated directly to machine code. It is needed to back-translate the SSA-form to normal form. In Chapter 3 a number of methods to do so are described.

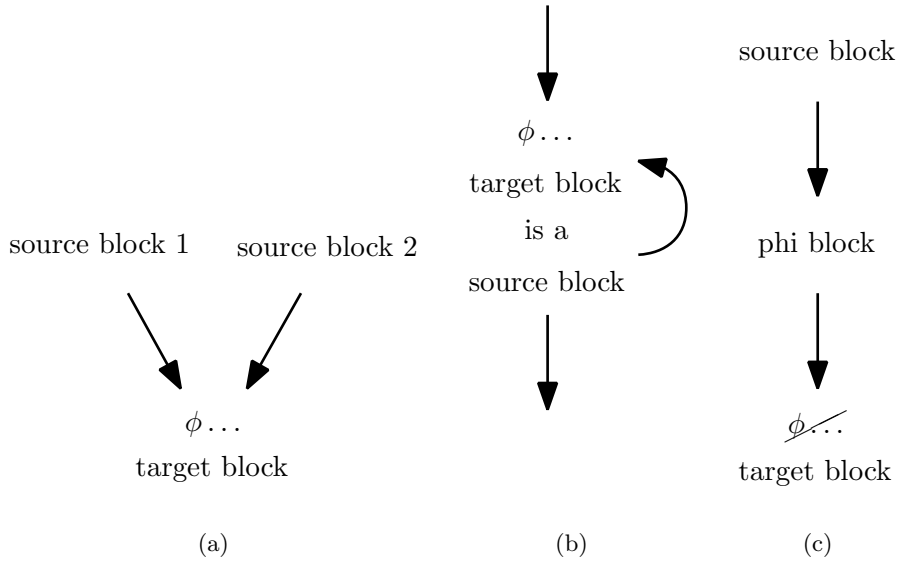


Figure 2.4: Various examples of source, target and phiblocks.

2.6 Block Names

Statements are placed in a basic block, a function is built from these blocks. In the rest of this work three different types of blocks in relation to a phi function are defined.

First of all, the block that contains the phi statement will be referred to as the *target block* (Figure 2.4a).

A phi function has a number of arguments. Each argument refers to a preceding basic block. These are referred to as *source blocks*. It is possible that the source block and the target block are one and the same block, for example in a loop (Figure 2.4b).

The third case is not found in SSA-form. But back-translation to normal form may introduce new basic blocks. Since such blocks are the result of a back-translated phi functions these blocks are referred to as phiblocks (Figure 2.4c).

2.7 Why SSA is Used

If a compiler uses SSA-form, the intermediate representation is translated into SSA, analysed and optimized and translated back out of SSA. The optimization passes need to ensure that the intermediate representation stays in SSA-form. This seems very complex and raises the question why SSA is used.

The dataflow of an IR in SSA-form is more explicit than the dataflow in normal form. Analysis can be done easier and faster, as Briggs[3] and Cytron[7] show. Here the small example of Figure 2.5 concerning dead code analysis is discussed. The first statement in this example is dead code, in normal form (Figure 2.5a) this can be detected by overwriting x without reading it. The SSA-form of the example (Figure 2.5b) is more explicit: x_1 is never used and can therefore be removed.

$$\begin{aligned} x &= 20 \\ x &= 40 \\ y &= x + 2 \end{aligned}$$

(a) Normal form.

$$\begin{aligned} x_1 &= 20 \\ x_2 &= 40 \\ y &= x_2 + 2 \end{aligned}$$

(b) SSA-form.

Figure 2.5: Example with dead code.

Chapter 3

Methods of Back-translation

The intermediate representation in SSA-form cannot be translated into assembly directly. It is needed to back-translate to normal-form first. This chapter describes different methods of back-translation in historical order.

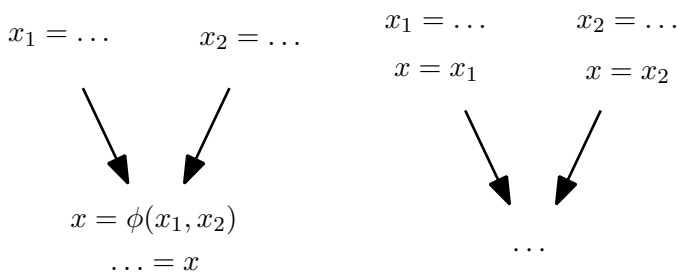
3.1 Cytron's Method

Cytron [7] was the first to publish about SSA form of intermediate representation. His paper acknowledges the need for a method of back-translation and also proposes such a method.

This method eliminates phi functions by replacing them with a number of copy instructions. For each argument to a phi function, a copy instruction is appended to the source block that copies the content of the argument to the target variable.

Figure 3.1 provides an example of Cytron's method of back-translation. In the example there is one phi function to which two arguments are passed. For both arguments, a copy statement is appended to the source block corresponding to that argument. The copy statement copies the content of the corresponding argument to target variable x. From this example it becomes clear that many new copy statements are created by back-translation. Some of these statements can be removed again by coalescing after back-translation, more details are provided in Section 3.3.

Besides using SSA-form to help analysis, it is also possible to use optimization techniques in SSA form. However the optimized SSA code can be more challenging to back-translate. Several years after Cytron's publication, Briggs [3] identified a number of problem-cases in which the SSA-form is changed in such a way that applying Cytron's method of back-translation fails to produce a result with the same meaning. These problem cases are created by transformations on the SSA-form, for example when copy propagation is applied to optimize the IR. Copy propagation tries to remove unnecessary copy statements, for example the right-hand-side of the second



(a) SSA-form

(b) Normal-form with ϕ-functions replaced.

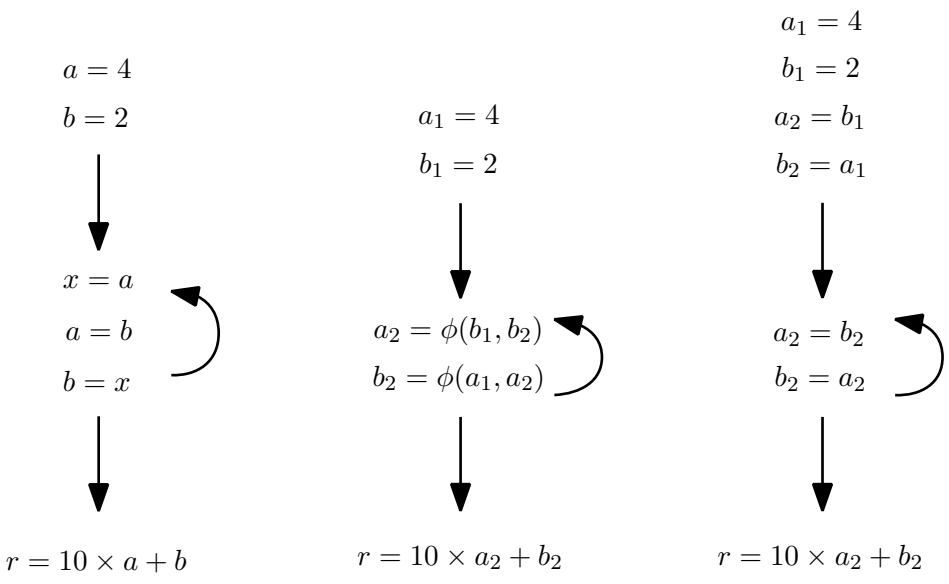
Figure 3.1: Example of applying Cytron's method of back-translation.

statement in $a = 40; b = a; c = b + 2$ can be removed when a is propagated: $a = 40; \cancel{b = a;} c = a + 2$.[1]

One of the problematic situations Briggs identified is known as the swap-problem. Figure 3.2 shows how this problematic situation is created, and how back-translating the copy-folded SSA form with Cytron's method goes wrong.

In this case, the error is in the central block. This block is both source and target. Cytron's method appends two copy statements to the end of the block. These statements should swap the variables a_2 and b_2. But the temporary variable that was used in the original is folded away by in-SSA optimization. In SSA-form (Figure 3.2b) this is correct since all phi-functions at the begin of a block are one simultaneous read-before-write action. But in the normal form that is the result of Cytron's method (Figure 3.2c) the value held by a_2 is overwritten before it can be stored in b_2.

[1] Yes, this example can be optimize further by also propagating the constant 40.



(a) Normal form before translation.

(b) SSA-form after translation and copy folding.

(c) Normal form after back-translation with Cytron's method.

Figure 3.2: Example code in which the swap problem manifests itself. Variable r in (a) and (b) is 24 or 42, depending on the number of iterations of the loop. But in Cytron's back-translated code, the content of a_2 is overwritten, therefore r in (c) is always 44.

3.2 Briggs' Method

Briggs [3] did not only identify the problems with Cytron's method but also proposed an improved method of back-translation. Like Cytron's method this method pushes out copy statements for all arguments in the phi-function. However, the target of this variable is a temporary variable (instead of the target of the phi-function itself.) The phi-function then, is not removed, but replaced by a copy from the temporary variable to the actual target. Figure 3.3 illustrates how this remedies the swap problem (see Figure 3.2-b for the SSA-form.)

Furthermore Briggs is also the first to suggest looking at liveness. He proposes an algorithm, like described above, but which emits (slightly) fewer copy operations. However, Sreedhar has taken this idea to a much higher level, as described in Section 3.4.

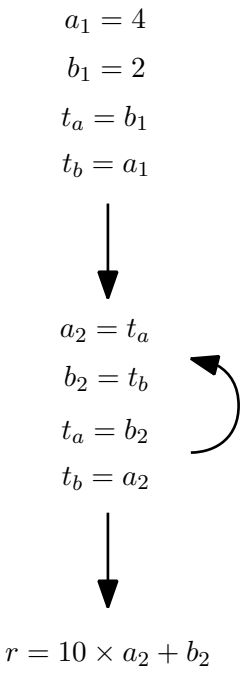


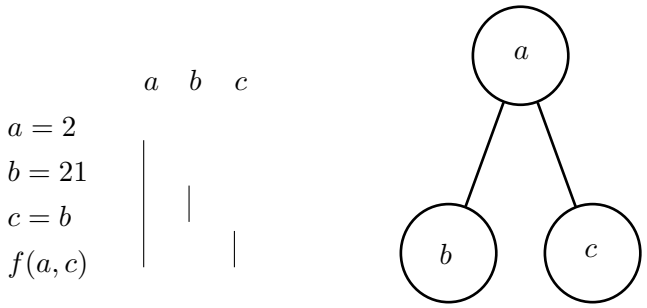
Figure 3.3: Normal form after back-translating optimized SSA-form of the swap problem using Briggs' method.

3.3 Post Back-translation Coalescing

A variable is live between the assignment of a value until the last use of that value. This is also called the liverange of a variable. In normal form a variable can have multiple liveranges. Since a variable in SSA is only defined (assigned to) once, it only has one liverange. Figure 3.4a shows an example with liveranges of the variables.

When the liveranges of two variables overlap, the variables interfere: they can not share the same register. Interference between variables can be represented with an interference graph. Each variable is represented as a node of the interference graph. If two variables interfere, there is an edge between the nodes (variables). An interference graph corresponding to the liveranges of the example is shown in Figure 3.4b.

If two variables do not interfere, it is possible to use a single variable. Merging variables is often called coalescing. The example interference graph shows that variable a and b cannot be coalesced since there is an edge between their nodes. There is no edge between b and c, therefore it is possible to coalesce these variables. After coalescing, the copy instruction $c = b$ is changes into a statement without effect and can be removed. The



(a) Example code. (b) Interference graph.

Figure 3.4: Liveness of example code and corresponding interference graph.

number of copy instruction in the example is reduced by coalescing c and b. Figure 3.5 shows that the result after coalescing variables b and c reduces the number of variables.

Many copy operations are introduced when applying Briggs' method of back-translation. The example of Figure 3.4 shows that copy operations can be eliminated by coalescing. Therefore, Briggs suggests to use Chaitin's method of coalescing[6], which uses the following steps:

1. Do a data-flow analysis to obtain liveness information.

2. Use this information to built an interference graph.

3. For all copy operations where source and target variable do not interfere, coalesce the variables and eliminate copy operation. This step is often called copy propagation.

4. Use a heuristic to colour the remaining interference graph.

5. If more colours than registers are used, add spill code and go back one step.

Other approaches are possible. A combination of coalescing and SSA back-translation is proposed by Sreedhar. He claims that applying his method is better than applying Briggs + Chaitin. His method is described in detail in Section 3.4.

Budimlic [5] proposes another combination of coalescing and back-translation. The result after applying his method is (according to his tests) comparable to Briggs + Chaitin. Budimlic's method uses SSA properties and therefore compile time is reduced. A drawback of this method compared to Briggs + Chaitin is that flexibility is lost, it is not possible to run other out-of-SSA optimizations before the coalescing phase.

$$a = 2$$
$$c = 21$$
$$f(a, c)$$

Figure 3.5: Variable b and c in the example of Figure 3.4 can be coalesced.

3.4 Sreedhar's Method

Sreedhar [13] tries to reduce the number of variables involved as well as the number of copy instructions needed compared to Briggs. The main idea of his method is to coalesce the variables involved in a ϕ-statement.

However, it is not always possible to coalesce the variables without changing the meaning. Sometimes variables interfere. In those cases the interference is to be broken first. This is done by pushing out copy-statements and introducing a new temporary variable. Then the argument in the phi function can be replaced by the new variable. Once interference is broken, the phi function contains a set of variables consisting of old and new variables. This set can be coalesced.

Sreedhar proposes three variations of his algorithm. The first variation assumes interference in all cases and is similar to Briggs' algorithm. Secondly liveness information is used to determine for which variables a copy instruction must be emitted. The final variant of Sreedhar's method uses both liveness information and knowledge on the dataflow to emit as few copy instructions as possible. Figure 3.6 provides an example on which Sreedhar's method is applied.

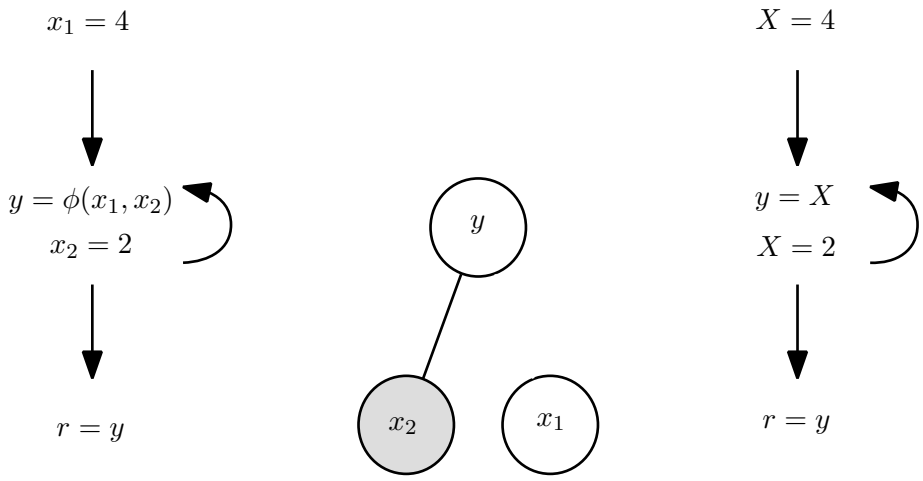
Combining liveness information with the knowledges from which source a phi function parameter originates is done for each set of two variables. If the variables in the set interfere, Sreedhar identifies 4 distinct cases. For each of these cases different measures are to be taken.

In the first and second case there is interference only on one of the sources, but not at the other. Interference can be broken by creating a single[2] copy instruction. This introduces a temporary variable.

In the third case, interference is identified at both sources. The only way to break the interference is to emit two copy instructions and introduce two new temporary variables.

The final case identified by Sreedhar is interference somewhere in the program between the two variables, but not at one of the source blocks. In this case a copy instruction is needed, but it does not matter to which of the source blocks this copy instruction is emitted. Therefore it is best to postpone this choice until copy instructions for all other argument pairs are

[2] As opposed to Briggs, where two copies would be emitted.



(a) SSA-form.

(b) Interference graph of variables involved in ϕ statement.

(c) Normal form after solving interference and coalescing.

Figure 3.6: Using Sreedhar's algorithm to back-translate the lost-copy problem. First a copy instruction is introduced to break interference between x_2 and y. To this end a new variable t (not shown in figures) is introduced that replaces the target and a copy instruction $y = t$ is placed just after the phi statement. Once interference is broken, all variables in the phi function are coalesced: $X \leftarrow \{x1, x2, t\}$ and the phi function is removed.

emitted. It is possible that a copy instruction to one of the arguments is already emitted because of interference with another variable involved in the phi function.

3.5 Boissinot's Improvements

Boissinot [2] studied the algorithms of Briggs and Sreedhar. Hereby he focussed on correctness of these algorithms.

He reports two cases in which back-translation with Sreedhar's method fails to produce a correct result. The first problem occurs when a rather exotic terminator is used in the IR. The second problem is about the liveness of variables at the end of a basic block. For both cases a solution is proposed by Boissinot, he also proposes the usage of an alternative interference check.

3.5.1 Exotic Terminator Problem

Generally an intermediate representation contains a limited number of statements for the control flow. For example goto or if-then-else. These state-

ments are only allowed at the end of a basic block, therefore they are called *terminators*.

Usually a terminator does not create nor alters any variable. Therefore it is safe to insert the copy operations just before the terminator as Briggs and Sreedhar propose.

However, Boissinot assumes an intermediate representation in which the terminating statement can also modify a variable. This is a common operations in digital signal processing, but modifying a variable is not SSA. However, Boissinot's almost-SSA form may be of some use where such statements are desirable. The example he uses is a branch-and-decrement statement, which is shown in the code snippet of figure 3.7a. Due to interference between variables, coalescing is not possible. A copy instruction would be inserted just before the branch-and-decrement statement. Therefore it would hold the wrong (not yet decremented) value of v.

Boissinot proposes two solutions for this problem. Either to split the edge and place the copy operation in a phiblock or to perform the same change as the terminator would apply in the copy operation. For the example of figure 3.7a, using $x' = x - 1$ as copy operations solves the problem.

3.5.2 Live Out Problem

Sreedhar tries to coalesce the variables of a phi function. For variables that interfere, his algorithm tries to emit as few copy operations as possible. To determine for which variables a copy operation is needed, the live-out sets at the end of the source blocks are used.

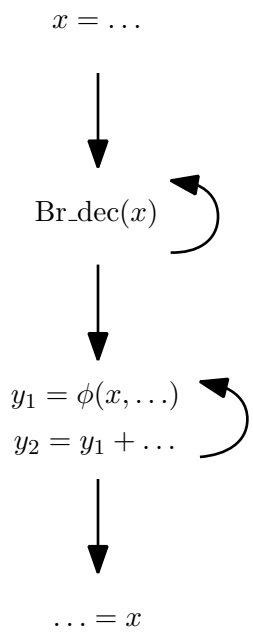
Boissinot sketches a situations where the live-out set of the block is not equal to the live-out set at the place the copy instructions are inserted. See figure 3.7b. The if-then-else terminator is the last use of y in that branch. Therefore y is not in the live-out set of that source block. If y is not in the block live-out set, interference can be broken by appending a copy instruction $tmp = x$. The result is shown in figure 3.7b.

However, the copy instruction is inserted just before the terminator at that point, y is in the live-out set. Interference is not broken and when y and tmp would be coalesed the meaning of the result `y=x; if (y)` deviates from the original.

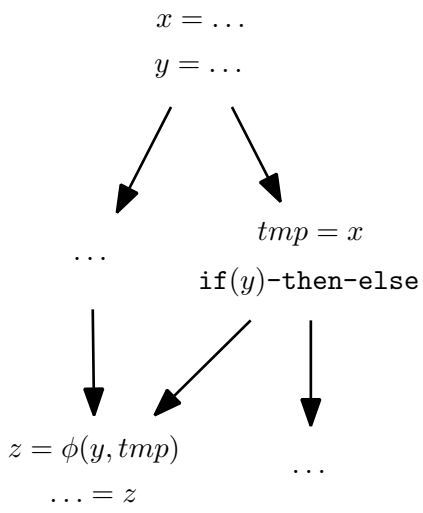
Boissinot's solution to the live-out problem is to look at the live-out set of the statement before the terminator of a source block instead of the live-out set of the source block itself, as Sreedhar suggested in his paper.

3.5.3 Improved Interference Check

Boissinot proposes to refine the definition of interference used in Sreedhar's algorithm. In SSA form, each variable has a uniquely defined value. Therefore Boissinot states that: two variables interfere if their live ranges overlap



(a) Exotic terminator



(b) Live-out problem after attempt to break interference with $tmp = x$ Coalescing y and tmp results in overwriting the variable used by the `if-then-else` statement.

Figure 3.7: Boissinot's problem cases.

and they hold different values. Sreedhar determines the interference of variables by checking if their live ranges intersect.

Boissinot counted the number of remaining copy operations in a number of benchmarks. He observes a reduction of the static number of copy operations in the results if he uses Sreedhar with value coalescing compared to Sreedhar with conventional coalescing. He does not report any information regarding execution time.

However, after copy propagation, there shouldn't be any copies left with the same value. Therefore this method is only valuable if the SSA form was not optimized. And in that case there are easier and faster ways to get out of SSA form.

Chapter 4

Proposal for Improvement

The methods of Briggs, Cytron and Sreedhar append copy instructions at the end of the source block. However a phi statement selects between a number of expressions *just before entry* of its target block. Therefore, insertion of a copy instruction at the edge between the source and target block would be closer to the definition of a phi function than appending it to the end of the source block.

The following sections elaborate on how this behaviour can be implemented, who used it before, what the effect on performance is, and provide suggestions to maximize performance.

4.1 Implementation

Adding copy instructions on an edge is not possible. However a new dedicated block for the copy instructions can be added. This phiblock is placed between the source and target block. Beside being a more natural way of placing the copy instructions, phiblocks also influence the execution time of the result.

4.2 Previous Work Mentioning Phi Blocks

One of the effects of using phiblocks is that the statements in the block are executed exactly in between the source and target block. Therefore Boissinot [2] suggests to use phiblocks as a possible solution for the exotic terminator problem, but only in case the source block is terminated with an terminator that changes a variable.

Budimlic [5] created an algorithm that provides a result similar to the combination of Briggs' method of back-translation and Chaitin's coalescing algorithm. In his paper Budimlic shows his result is almost as good as Briggs and coalescing. But the compiler itself is faster as it uses the properties of SSA when coalescing. Budimlic's algorithm cannot handle critical edges.

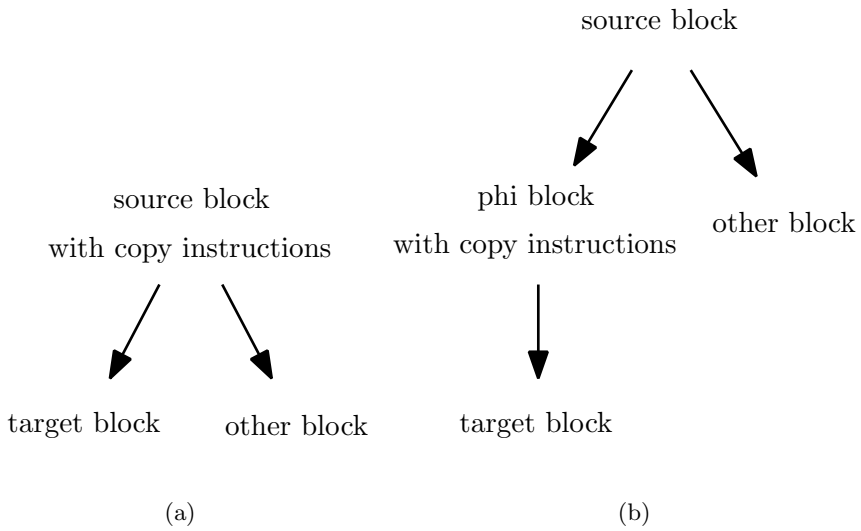


Figure 4.1: The number of of executed copy operations introduced by SSA back-translation can be reduced with the insertion of a phiblock.

That is, an edge for which the source has multiple outgoing edges and the target multiple incoming edges. Before back-translation Budimlic splits all critical edges by inserting a basic block.

As far as known by the author, there is no work about the effects of phiblocks on the performance of the end result produced by the compiler.

4.3 Conjectures on Performance

Budimlic and Boissinot suggest the use of phiblocks are required in certain cases to obtain a correct result.

In this document the positive effect on the performance of the resulting code is discussed. It is likely the dynamic number of copy operations decreases (especially in combination with post optimization) when using phiblocks; on the downside a number of extra unconditional jump operations are needed for the phiblock insertion.

It is expected that number of executed copy operations decrease, because the copy operations are only executed when really needed. This is illustrated by the example of Figure 4.1. The copy instructions needed by the target block are in case a always executed. For case b, they are executed only when necessary.

The inserted phiblocks are only executed when traversing exactly from the intended source- and targetblock. Therefore it is expected that the number of jump operations increases. Figure 4.2 provides an example that

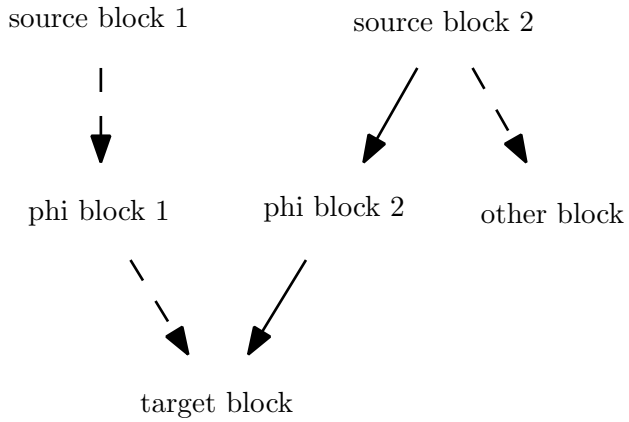


Figure 4.2: Introduced phiblocks can increase the number of jumps needed. Dashed edges can be implemented by falling through.

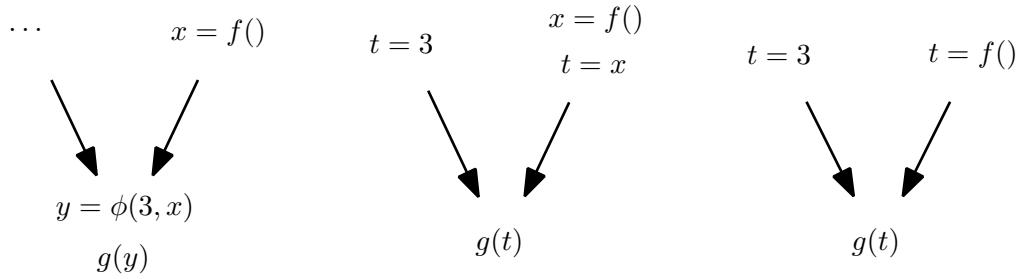
shows that the insertion of a phiblock can introduce the need for an extra unconditional jump (phiblock 2 introduces 1 extra jump operation). But that this is not always the case (phiblock 1).

4.4 Post Back-translation Optimizations

Section 3.3 discussed that after Briggs' method of back-translation, it is good practice to coalesce variables by copy propagation (Chaitin's method). However there are situations for which copy propagation is not enough, for example, if not only variables but also constants or even more complex expressions are used as argument to a phi function.

In Figure 4.3 an example is shown where after back-translation a combination of copy propagation and expression propagation is able to reduce the number of copy operations compared to copy propagation alone. In this example Briggs' method is used to back-translate from SSA form, which introduces copy instructions that are unnecessary.

Splitting edges with phiblocks will have a beneficial effect on the result of optimizations that run after back-translation since not only the execution of statements is more selective but the live ranges of the variables involved in these copy instructions also will be uncluttered.



(a) SSA-form.

(b) Back-translation and copy propagation.

(c) Back-translation, copy propagation and expression propagation.

Figure 4.3: An example where copy propagation is not enough. It is not possible to propagate x from the copy statement $t = x$. However the expression holding function call $f()$ can be propagated. Then x is never used and therefore $x = f()$ is dead code and can be removed.

Chapter 5

Method of Evaluation

To obtain empricial data on the effect of phiblocks insertion during back-translation Sassa's evaluation is used as a foundation. Section 5.1 summarizes his method and results. The method used for this research is described in Section 5.2

5.1 Sassa's Evaluation

Sassa [12] evaluated the back-translation methods proposed by Briggs and Sreedhar. He performed an analysis of the algorithms on typical problem cases. And he ran a series of benchmarks to obtain empirical data. Based on his results Sassa concluded that Sreedhar's algorithm is superior to that of Briggs. The next subsections summarize Sassa's experiments.

5.1.1 Preliminary Comparison

Briggs identified three problem cases on which Cytron's back-translation method fails. Sassa assumes these cases are typical for an optimized program in SSA form. Therefore he bases his preliminary comparison on these cases.

Sassa back-translates each of these problem cases with both algorithms. He tries to coalesce the remaining variables and removes dead code.

The number of copy operations left is used as the metric to estimate performance. Table 5.1 shows Sassa's result. Sreedhar's method results in a better result on the Swap problem. For the other two cases the number of copy operations is equal for Briggs and Sreedhar.

5.1.2 Empirical Data

To support his conjecture Sassa did an experiment in which he tried to simulate real world compilation. For this he used 6 integer benchmarks from the SPEC 2000 suite. To compile the benchmarks he used the COINS compiler. In this compiler Sassa implemented both back-translation algorithms. To

	Briggs	Sreedhar
Lost Copy	1	1
Simple Ordering	2	2
Swap	5	3

Table 5.1: Copy operations in result as counted by Sassa. Fewer is better.

obtain results applicable to both compilers for general purpose computing as well as the embedded market, the experiments are performed with 20 and with 8 registers.

Sassa sees the biggest difference between Sreedhar and Briggs in the gzip benchmark: With 8 registers, the execution time of the benchmark compiled with Sreedhar's method is about 28% faster than the result compiled with Briggs' method. With 20 registers, the difference is 8%.

For the other testcases from this suite, the difference is smaller. With 8 registers, Sreedhar is always faster than Briggs. With the number of registers increased to 20 it is still true that Sreedhar is faster on average. However, there are also two cases in which Briggs is faster. The result of the mcf benchmark on which Briggs' method is applied is faster by almost 3% than Sreedhar's result. That is the largest difference in performance in favor of Briggs.

5.2 Method Used for this Study

In this study we want to evaluate the effect of post back-translation optimization and the effect of phiblocks. First a preliminary comparison is performed based on typical cases, then empirical data is obtained with experiments.

In his preliminary comparison, Sassa observes the biggest difference between Brigg and Sreedhar in the swap problem. Therefore I will do a case study of the swap problem to find out what the effect of post back-translation optimization is and what the effect of phiblocks is for this case.

Also, in an example I study the relation between the number of variables and the number of registers / spills needed. To collect empirical data I use the two cases in which Sassa finds the two most outspoken differences: mcf and gzip from the SPEC2000 suite.

To compile the benchmarks I used a combination of two compilers. The front-end used is from the LLVM compiler. This part translates the C-code to the IR of LLVM in SSA form. The in-SSA optimization are done by this compiler. I wrote a translator from LLVMIR to CCMIR in SSA form. On the result one of the following methods of back-translation is performed:

- The original method of Briggs, which appends copy instructions to the source block.

- The original interference and dataflow method of Sreedhar, which appends copy instructions –needed to break interference– to the source block.
- Briggs' method with modification, inserts and appends to phiblocks.
- Sreedhar's method with modification, inserts and appends to phiblocks when needed to break interference.

After back-translation a number of existing CoSy optimizations can be performed on the result. The CoSy compiler generates assembly from which an executable is created. The executables of various combinations of back-translation with or without post back-translation optimization is run on a pentium machine to obtain execution times.

Chapter 6

Case Study

I studied two cases. The first case discussed is the swap problem a typical problem case that is encountered by a back-translation algorithm. It is used to show the differences between the backtranslation methods of Sreedhar and Briggs, the effects of phiblock creation and the effect of optimization after back-translation. Also the trade-offs of edge-splitting are discussed in this section.

Section 6.2 discusses the order in which optimizations are done. Sreedhar coalesces variables in combination with back-translating. Therefore, the number of variables in the result of back-translation with Sreedhar is often lower than the number of variables after using Briggs' method. However the early coalescing of Sreedhar may actually result in a larger number of registers needed.

6.1 Swap Problem

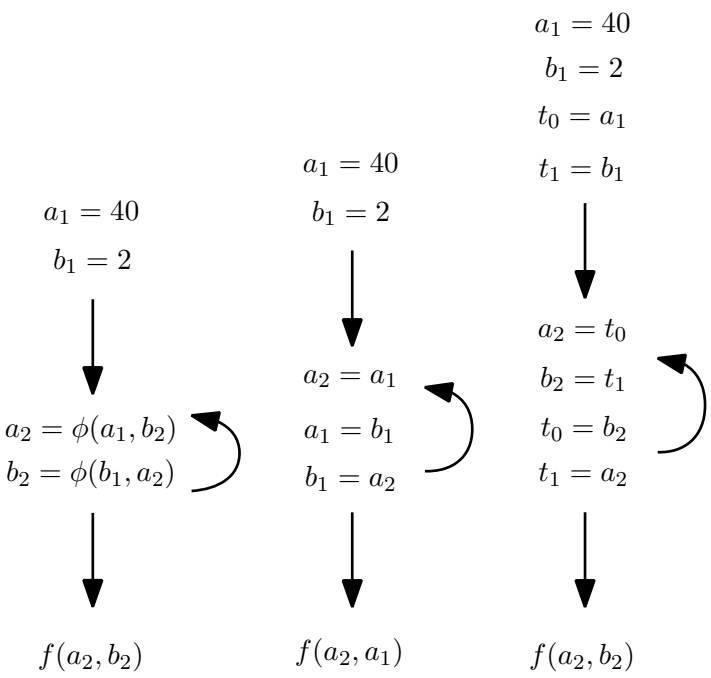
The swap problem is a typical case for which Sreedhar's original method of back-translation performs better than Briggs' orignal method. The swap problem (Figure 6.1a) is one of the 3 problem cases identified by Briggs for which Cytron's method fails (see Chapter 3.) This section takes a closer look at why Sreedhar performs better than Briggs in this case. Also the effect of the phiblock modification is studied for this testcase.

6.1.1 Result of Sreedhar's Method

Coalescing the variables involved in the phi functions is not possible immediately. Sreedhar's method therefore emits a number of copy instructions to break interference.

The second phi statement is identified as case 2 interference[1] between

[1]Section 3.4 explains the different cases of dataflow and interference recognized by Sreedhar's method of back-translation.



(a) SSA-form (b) Sreedhar (c) Briggs

Figure 6.1: Swap problem back-translated with unmodified Briggs and Sreedhar before any kind of post back-translation optimization is applied.

source and target. Only one copy is needed. The first phi statement then needs two copies to break a case 3 interference between source and target. Figure 6.1b shows the result.

6.1.2 Result of Briggs' Method

Briggs' method introduces copy operations for each argument of the phi-function at the end of the source block. There are two phi functions in the swap problem of Figure 6.1a with two arguments each, thus Briggs' method emits 4 copy operations. Furthermore each phi-function has a target variable to which the temporal variable is to be copied, resulting in two more copy operations.

Of these 6 copy operations 4 are inside the body of the loop and two are outside the loop. The two copy operations outside the loop can be removed by propagation but because of overlapping live ranges in the body, the number of copy operations inside the loop cannot be reduced.

method	copy instructions	branch instructions
Briggs	6(4,4)	1(1)
+ propagate	4(4,4)	1(1)
+ phiblocks & propagate	3(3,1)	2(1)
Sreedhar	3(3,3)	1(1)
+ propagate	3(3,3)	1(1)
+ phiblocks & propagate	3(3,2)	2(1)

Table 6.1: Number of instructions after back-translation. Figures in parentheses show number of instructions in loop body. Before the comma if more iterations follow. After the comma the number of instructions executed in the last iteration of the loop.

6.1.3 Effect of Phiblock Insertion

Figure 6.2 presents the result of appling the method of Briggs with phiblock insertion on the swapproblem. Two phiblocks are introduced, one block in the loop and one block before entering the loop.

Like unmodified Briggs, the 2 copy operations before entering the loop can be removed by applying a combination of expression and copy propagation. That leaves the phiblock empty, so it can be removed.

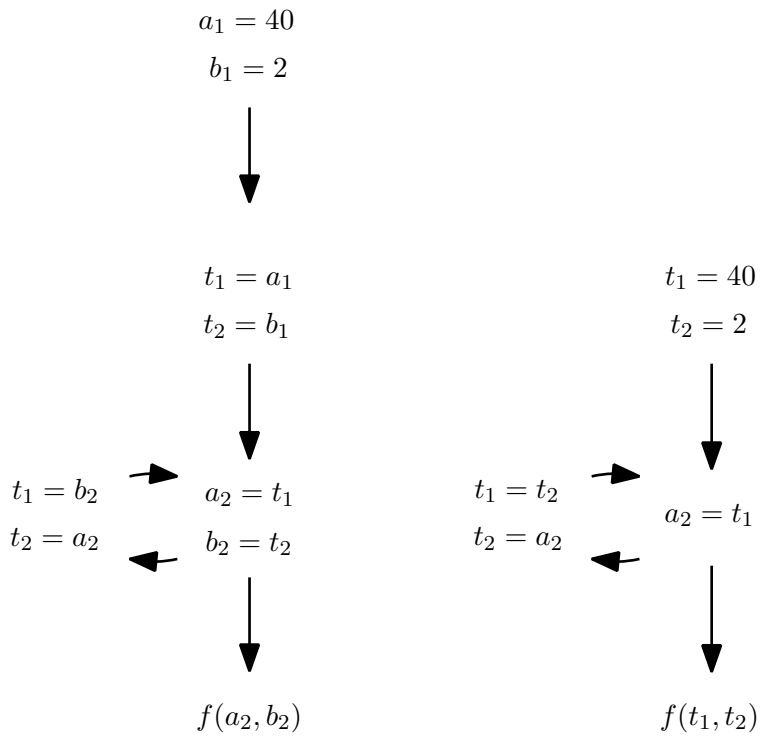
The phiblock inside the loop reduces the length of the live ranges of t_0 and t_1. Therefore the number of copy operations inside the loop can be reduced from 4 to 3 by applying copy propagation.

Also the last iterations of the loop does not enter the phiblock. Thus, the last iterations only executes one copy operation. The trade-off is that on entering the loop an unconditional branch instruction is needed to jump around the phiblock.

Modified Sreedhar does not suffer from the unnecessary long live-ranges. Therefore the effect of the phiblocks is smaller. The body contains two copy operations and the phiblock one. Therefore the last iteration executes two copy instructions. Thus, the number of copy operations is reduced by one, but one branch instruction is introduced.

6.1.4 Conclusion

An overview of the number of copy and branch instructions of the various methods is provided in Table 6.1. Comparing the results of the backtranslated swapproblem without modification nor optimizations show a better result when Sreedhar is used to backtranslate than when Briggs is used, because the number of dynamic copy operations is lower and the number of branches is equal.



(a) modified-Briggs (b) + copy and expression propagation

Figure 6.2: Swap problem back-translated with modified-Briggs and the effect of optimizing after back-translation.

Applying propagation to these back-translation results reduces the gap between Briggs and Sreedhar. However, the number of dynamic copy operations in the result of Sreedhar's method is still lower even though Sreedhar's result does not profit at all from these optimizations.

Splitting edges by creating phiblocks reduces the dynamic number of copy operations. However, both the static and dynamic number of branch instructions increases. For Briggs, a reduction of two copy instructions is traded for one extra branch instruction. Sreedhar with phiblocks compared to Sreedhar without trades one copy instruction for one branch instruction.

The introduced branch instruction is an unconditional jump, to a location nearby. The cost of such a jump is very low on modern processor architectures. Since the address is known, the pipeline is not disturbed. Many processors even implement a technique called branch folding [9], this virtually reduces the cost of this jump to zero.

From Table 6.1 can be concluded that Briggs with phiblocks is faster than plain Sreedhar if the cost of an unconditional branch instruction is equal to (or less than) the cost of a copy instruction.

It is very unlikely that modified Briggs is slower than the fastest unmodified result. Only when an unconditional branch would be 3 or more times as expensive as a copy instruction that holds.

6.2 Less is More

The number of variables in the intermediate representation is often smaller when using Sreedhar's method compared to using Briggs' method. This is caused by the coalescing technique used in Sreedhar. However, Hack [8] showed that aggressive optimization in an early stage may have a detrimental effect on coalescing in the final stage of the compiler, resulting in the need for a larger number of registers. In this section an example is studied showing that Sreedhar's coalescing indeed can result in fewer variables for which more registers are needed compared to the result of Briggs' method.

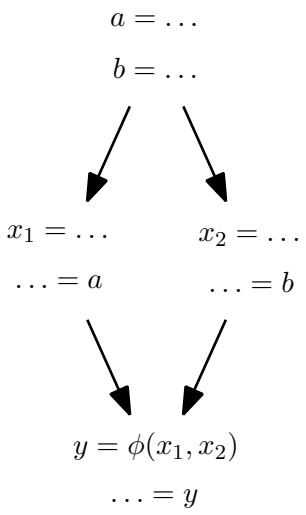


Figure 6.3: SSA form of the example

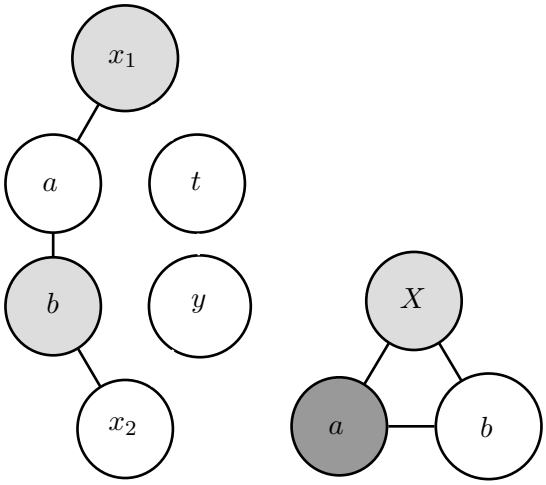
In this example a phi-statement is used in which none of the variables involved (x_1, x_2 and y) interference. Ideal for Sreedhar. However, there are two other variables (a and b) which interfere with each other and with one of the arguments to the phi-function. See figure 6.3 for the SSA-form of the example case.

The testcase is back-translated with the two different methods, of which the results are shown in Figure 6.5. The intermediate representation of

Briggs needs 6 variables, that is the 5 original variables plus a new one. Sreedhar coalesces x_1, x_2 and y, resulting in 3 variables. Briggs' result needs 10 copy instructions, compared to 7 used in Sreedhar's result.

So far, Sreedhar seems to be better. Let us, however, take a look at the interference graphs of the resulting intermediate representations. Also here we see the difference in number of variables reflected. However, these interference graphs allow us to determine the number of registers needed. This can be done with graph-coloring. Every different colour is a different register. The minimum number of colours needed for Briggs' graph is two. Sreedhar's graph has only 3 variables, but all 3 of them need a different colour. Thus, in this case, Briggs' result has need for fewer registers than Sreedhar's result.

When mapping to registers, a number of copy instructions of Briggs' intermediate representation become superfluous. They copy the content of a register to itself. By removing these instructions, the number of copy instructions in Briggs' result is reduced to the same number of instructions as Sreedhar's result.

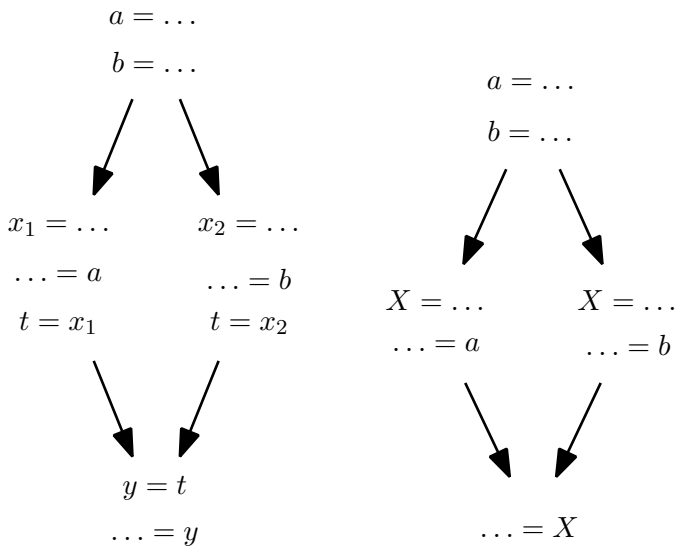


(a) Briggs (b) Sreedhar

Figure 6.4: Interference graphs.

6.2.1 Conclusion

In this section an example is presented showing that while the coalescing of Sreedhar's method of back-translation reduces the number of variables it also may block coalescing in a later phase. In certain cases this can result in the need for a larger number of registers after Sreedhar's method compared to the result after applying Briggs' method.



(a) Briggs (b) Sreedhar

Figure 6.5: Back-translated normal-form.

Chapter 7

Implementation

The quality of the implementation of an algorithm depends on the craftsmanship of the programmer and the quality of the design. This chapter starts with a description of what was implemented for the experiments, followed by a discussion of how difficult it is to create a high quality implementation of the back-translation algorithms of Sreedhar, Briggs and the edge splitting modification of those algorithms.

No hard, proven metric known to the author of this thesis in which to express the quality of a program based on the algorithm. However, Budgen [4] describes a set of factors to assess the fitness for purpose, which are applied in this chapter to the (modified) algorithms of Sreedhar and Briggs. Before the fitness for purpose can be discussed, it is needed to define the objectives of the method of back-translation, which is done in the first section of the chapter. Then complexity, reliability, efficiency and testability are discussed.

7.1 Implemented for this Study

For the experiments in this study certain parts from the LLVM compiler and CoSy's framework were readily available, other parts were implemented as part of the thesis work. Figure 7.1 shows all components of the compiler chain.

A parser was created that reads LLVMIR and maps it onto SSA-extended CCMIR. To back-translate the SSA-extended CCMIR, the methods of Briggs and Sreedhar were implemented, with and without the edge-splitting variation proposed in Chapter 4.

7.2 Complexity

Complexity is an indicator of the of the likeliness an error is made in the implementation. Various metrics exist to estimate the complexity of an im-

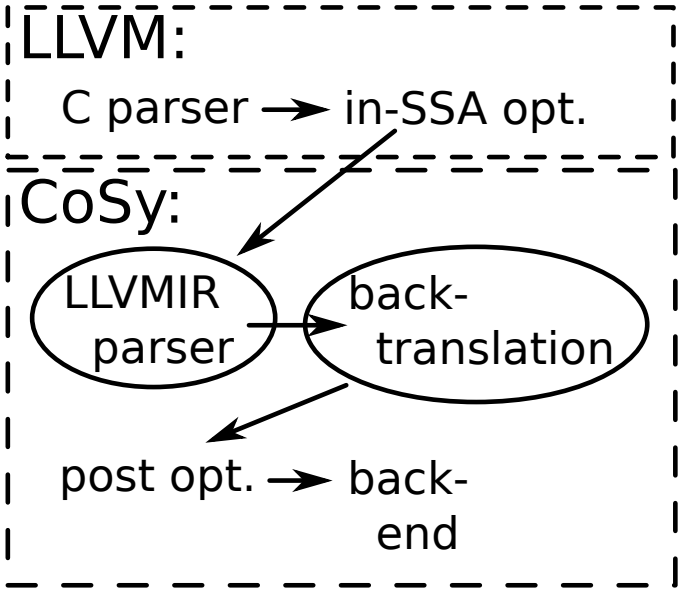


Figure 7.1: Compiler chain with new components encircled.

plementation. In this section three of these metrics are used to compare the methods of Sreedhar and Briggs, and to estimate the increased complexity of edge splitting. The metrics used are lines of code, number of branch statements and number of loops.

Table 7.1 shows that the implementation of Sreedhar's method is estimated to be 9 to 37 times as complex as the implementation of Briggs' method. Liveness analysis needed by Sreedhar is excluded in this analysis. Phiblock insertion is estimated to be no more than 2 times as complex as appending to sourceblocks, as can be seen in Table 7.2.

	Sreedhar	Briggs	ratio
lines	245	28	9
branches	37	1	37
loops	17	2	9

Table 7.1: Indicators of the complexity of implementation of the back-translation methods of Sreedhar and Briggs.

7.3 Reliability

The two factors to estimate the reliability of back-translation based on the design are completeness and robustness.

	phiblock	sourceblock	ratio
lines	75	40	2
branches	11	6	2
loops	9	7	1

Table 7.2: Indicators of the complexity of implementation of copy instruction appending to sourceblock or edge-splitting appending copy instructions to source compared to inserting in a phiblock.

Completeness is the ability to handle all possible inputs. In the case of a back-translation algorithm, variations in input come from the arguments to the phi statements. By its generic nature, Briggs can handle almost any possible expression as argument. The more specific design of Sreedhar requires a local variable as argument to the phi function.

Robustness is the ability to cope with failure of other components. Sreedhar depends on the liveness information, a bug in the liveness analysis can result in subtle but serious deviations in the result. On the other hand, Briggs does not rely on liveness information, but coalescing after back-translation of course does.

7.4 Efficiency

Two approaches are possible. One is to analyze the algorithms, the other approach is to measure the time needed by the compiler.

Simplified algorithms of Briggs and Sreedhar are shown in Listing 7.1 and Listing 7.2. For the analysis I assume the phi-statement to back-translate has $n-1$ arguments and 1 target. Briggs emits a copy for each argument and for its target, thus it needs $O(n)$ time. Sreedhar compares each pair, needing $O(\binom{n}{2}) = O(\frac{n(n-1)(n-2)!}{2(n-2)!}) = O(\frac{1}{2}n^2 - \frac{1}{2}n) = O(n^2)$ time.

Thus from this can be concluded that the time needed by Briggs grows linearly with the number of arguments whereas the time needed by Sreedhar grows polynomial. However, as the number of argument is small (n is typical between 3 and 5) constants are of a greater importance and the conclusion based on big-oh analysis should be used with care.

Listing 7.1: Big-Oh analysis of Briggs' algorithm

```
create tmp
for each obj in phiStatement->args                    O(n)
        emit copy instruction tmp = obj
emit copy instruction tar = tmp
```

Listing 7.2: Big-Oh analysis of Sreedhar's algorithm

```
for each pair(i-j) in args + target                    O(C(n,2))
        check if i is in Lj
        check if j is in Li
        based on interference
                add i, j or both to list of candidates
for all objects in candidates                          O_wc(n)
        emit appropiate copy instructions
```

The comparison of the compilation time of the implementation of Sreedhar, Briggs and the phiblock modification is done by measuring the time needed to compile the testcases Figure 7.2 shows a comparison of the obtained times.

Comparing the time needed by the phiblock modification of Briggs' algorithm with the time needed by the unmodified Brigg's algorithm shows a small increase in time needed to insert the phiblocks.

The time needed by Sreedhar's algorithm to complete back-translation depends on the live-ranges of the variables is involved. Since the phiblock modification influences the liveranges of the variables, the impact of the phiblock modification on the time needed by Sreedhar's method to back-translate is larger than on Briggs.

Comparing Sreedhar's algorithm with Briggs algorithm shows that with Sreedhar's method the compiler takes up to a factor 10 more time than when Brigg's method is used. However, in the implementation created during this thesis the liveness analysis does not use the SSA properties. And more important, updating the liveness information in the implementation of Sreedhar's algorithm can be done more efficient, as described by Sreedhar in his paper. Therefore comparing compilation times of Sreedhar and Briggs based on these measurements is not completely fair. Nonetheless variations of compilation time of the phiblock modification are a good indicator that Sreedhar is slower than Briggs although the inefficiency in the liveness analysis amplifies the differences of the compilation time for Sreedhar's algorithm.

7.5 Testability

The action for an argument of a phi function in Brigg's algorithm is always the same. Per pair of of variables Sreedhar distinguishes 4 different cases when there is a conflict between the variables. For each of these cases, and the case in which there is no conflict a different course of action is to be taken. A phi function with $n-1$ arguments and 1 target variable therefore has $\binom{n}{2} \times 5$

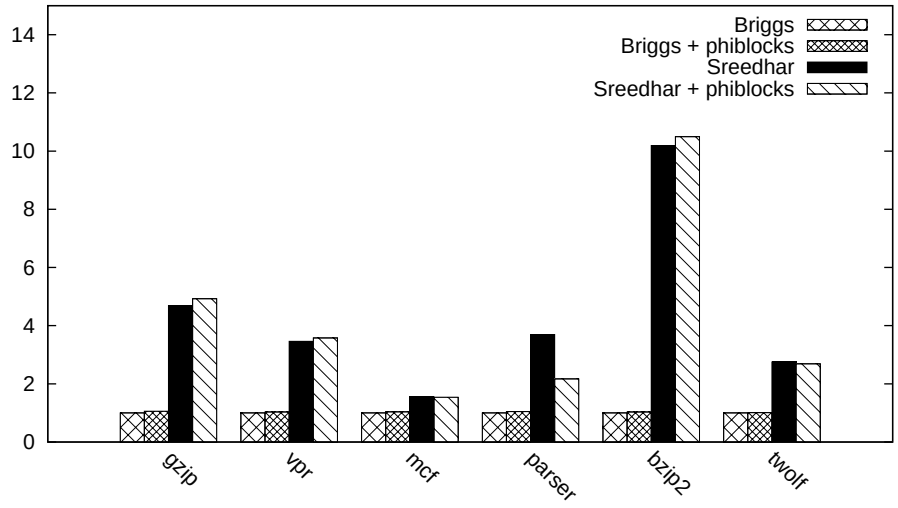


Figure 7.2: Six testcases from the SPEC benchmark suite are used to obtain compilation time ratio per benchmark when method of back-translation is varied, normalized to Briggs without phiblocks. In the next chapter more details on the benchmarks follows.

possible resulting variations. For 3 arguments that are 192 possible results! It is therefore much more difficult to *test* an implementation of Sreedhar's method than to test Brigg's method.

Sreedhar's method depends on liveness information that is to be created before the algorithm is run and also needs to be updated in between back-translation of the phi statements. A small programming mistake in either the live analysis before or during back-translation with Sreedhar can result in bugs that are hard to find.

Coalescing variables make it harder to verify if an erroneous back-translation is made. In the case of Briggs + Chaitin, a dump can be made in between, or Chaitin can be disabled completely. The variable coalescing in Sreedhar's algorithm is part of the back-translation and cannot be skipped.

Spliting edges does not make it easier nor harder to test correctness of the implementation. However, in the case of Sreedhar's algorithm they do change the behaviour of the algorithm as the (intermediate) live ranges of the variables differ compared to those when back-translating with Sreedhar's original method of back-translation.

Chapter 8

Results

This chapter presents the results of 144 experiments in which 6 testcases were back-translated with 4 methods of back-translation and optimized with 6 different combinations of engines.

The first section of this chapter explains the setup of the experiments. In the other sections the results are discussed. In Section 8.2 it is shown what the effect on the performance of the result is when phiblocks are or are not created during back-translation. Section 8.3 discusses the differences between Sreedhar and Briggs. The effects of post optimization are discussed in Section 8.4. For reference Table 8.1 lists the execution times of all benchmarks.

8.1 Experiment Setup

To obtain empirical data the result after compilation of various testcases is executed. This section starts with a description of the testcases that are used. Then follows an explanation of the various optimization schemes. The last part of this section shortly sums up the methods of back-translation benchmarked.

8.1.1 Benchmarked Testcases

A selection of 6 testcases of the SPEC 2000 integer benchmark suite are used: gzip, vpr, mcf, parser, bzip2 and twolf. These testcases are selected because they are used in previous research of Sassa [12]. This enables comparison with his results. Refer to Chapter 5 for more details on Sassa's evaluation.

Gzip and bzip2 are testcases that compress and decompress. Vpr and Twolf find solutions for the place and route problem. Parser is a syntactic parser of a natural language. Mcf is an implementation of the simplex algorithm to optimize a schedule.

For a 20 register machine, Sassa found results back-translated with Sreedhar performing on average better than those back-translated with Briggs. Outlyers are gzip for which Sreedhar's result is faster by 8% and mcf for which the Briggs' back-translated result is faster by almost 3%.

During the experiments, the bzip2 testcase triggered strange behaviour in the compiler. To obtain a valid result in all cases; it was necessary to the disable the loop invariant optimization engine. An error is suspected in the creation of liveness information for Sreedhar. Therefore results of bzip2 should be used with care.

8.1.2 Optimization Schemes

Before back-translation, the SSA code is optimized by the LLVM compiler (among others, this includes expression propagation which is the root cause of Cytron's method failing to produce a correct back-translations result). The in-SSA optimization techniques are for all benchmarks the same (LLVM -O3). Various combinations of optimizations in CoSy on the normal form intermediate representation after SSA back-translation are tried. The different combinations are described here shortly:

noopt
: The back-translated intermediate representation is passed to the codegenerator immediately.

minopt
: Block merging, dead code removal, disjoint life range splitting, expression simplification and miscellaneous small optimizations.

constprop
: Search for assignment of a constant to a variable and replace use of that variable with the constant. Includes minopt.

copyprop
: Search for copy statements and try to replace use of left-hand-side variable with right-hand-side variable. Includes minopt. This basically is Chaitin's coalescing algorithm.

exprprop
: Search for an assignment of an expression to a variable and try to replace use of that variable with the right-hand-side expression. Includes minopt.

maxopt
: Minopt and all 3 propagation techniques. This is the default combination of CoSy's low-level optimization techniques.

8.1.3 Method of Backtranslation

The in-SSA intermediate representation of the benchmarks is back-translated with 4 different methods of back-translation: Briggs' method, Sreedhar's method and both methods with phiblock insertion. In the following the term edge-splitting is used synonymously with phiblock insertion.

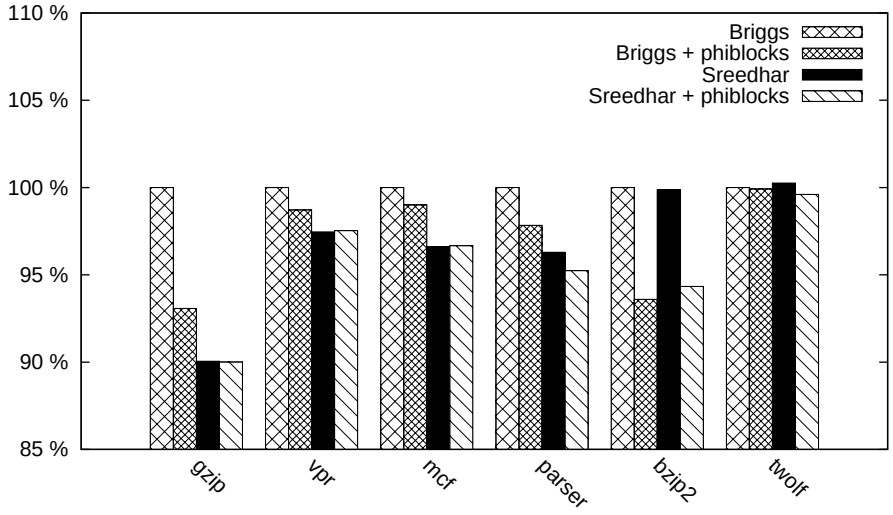


Figure 8.1: Relative execution times of the results (lower is better), when the IR after back-translation is directly fed into the codegenerator (noopt). Per benchmark the execution times are normalized to the execution time of the result back-translated with Briggs' method.

8.2 Sreedhar, Briggs and Edge-Splitting

Two comparisons of the effect of edge splitting by phiblock insertion on the algorithms are presented here. First the execution times are compared when no optimization is applied. Then for each method the various post optimizations schemes as described above are applied and the best result is selected.

Figure 8.1 shows the execution times of the resulting executables when no post back-translation optimization is applied after back-translation. For all 6 testcases, the result after applying Briggs with phiblocks is faster than the result on which Briggs without phiblocks was used to back-translate. The result of Sreedhar with phiblocks is faster in all but bzip2.

Optimization is beneficial after back-translation with phiblocks and without phiblocks, details are discussed in Subsection 8.4. In the bar-chart of Figure 8.2 the methods of backtranslation are compared when post-backtranslation optimization is used. Briggs' method with phiblocks is faster than Briggs

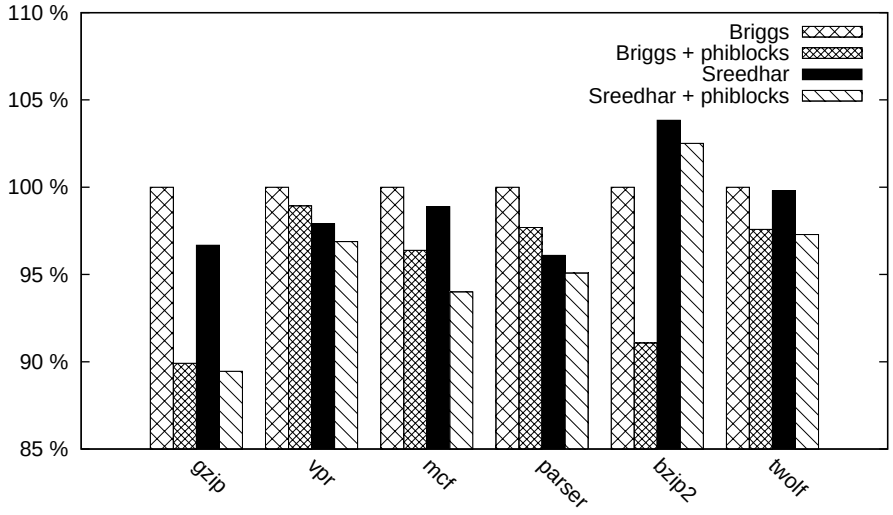


Figure 8.2: Comparison of the best result of each back-translation method. For each benchmark the fastest result after applying Briggs (without phiblocks) is set to 1. The other results are scaled to to that result.

without phiblocks in all cases, on average the phiblock modification is 5% faster. Sreedhar with phiblocks also gives a faster result compared to Sreedhar without phiblocks; on average execution time can be reduced with 3%.

8.3 Sreedhar versus Briggs

Sassa's conclusion that Sreedhar is superior to Briggs is confirmed by the results of back-translation with optimization and without phiblocks in all cases but bzip2. In the cases of gzip, mcf, bzip2 and twolf the result of Briggs with phiblocks is faster than the result for which Sreedhar without phiblocks is used. However, in 3 of those 4 testcases Sreedhar with phiblocks is still faster than Briggs with phiblocks.

Figure 8.3 shows for each testcase of the benchmark suite, the relative difference in execution time of the result after applying either Sreedhar or Briggs. In most cases the largest difference between Sreedhar and Briggs is shown when no phiblocks are used. This is especially pronounced in the testcase gzip. For this testcase the result of Sreedhar without modification and without applying post optimization executes in 90% of the time needed by the result of Briggs without modification and post optimization. When phiblocks are used during back-translation and optimization thereafter is applied, then Sreedhar's result executes in 99% of the time needed by Briggs' result.

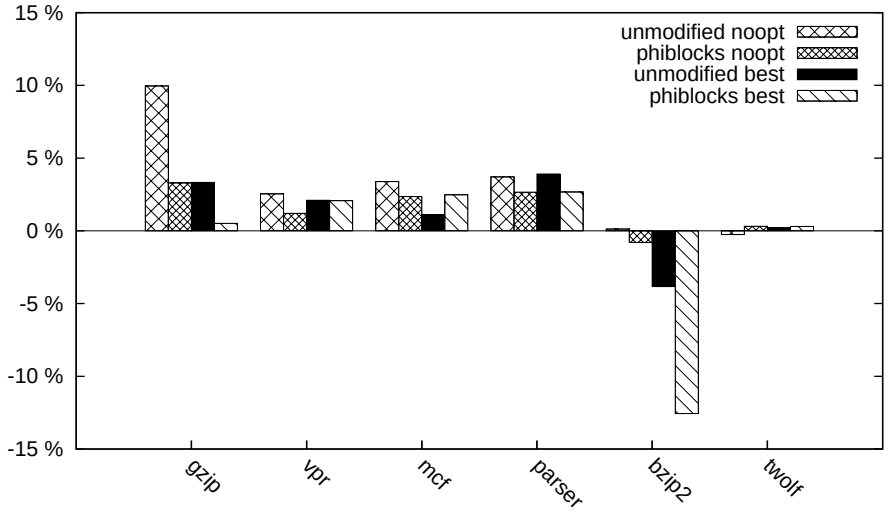


Figure 8.3: Difference in execution time when Briggs or Sreedhar is used with and without phiblock modification.

8.4 The Effect of Optimizations

In this section a closer look is taken at the effects of the post back-translation optimizations. First the gain of the best results over the results where no optimization was applied, then a closer look is taken at the differences between the various optimization schemes.

Figure 8.4 shows how much is gained by post optimization for the 4 methods of back-translation. Most interesting aspect of this chart is the difference between optimizing after Sreedhar with and without phiblocks. For the 4 cases gzip, vpr mcf and twolf, optimizing after Sreedhar with phiblocks is more beneficial than optimizing after Sreedhar without phiblocks. These cases are those for which phiblocks seemingly had no effect in the noopt case (see Figure 8.1). On average post optimization gains an improvement over noopt of 17% when Briggs is used, 18% after Briggs with phiblocks, also 18% after Sreedhar and 20% after Sreedhar with phiblocks.

The positive effect of phiblocks for Briggs 1998's method is already visible in the noopt benchmarks and the effect of post optimizations is not univocal more beneficial when Briggs with phiblocks is used than when Briggs without is used.

Figure 8.5 and Figure 8.6 show for all 6 testcases a barchart with the scaled execution times of the results after applying the various optimization schemes for the 4 different methods of back-translation.

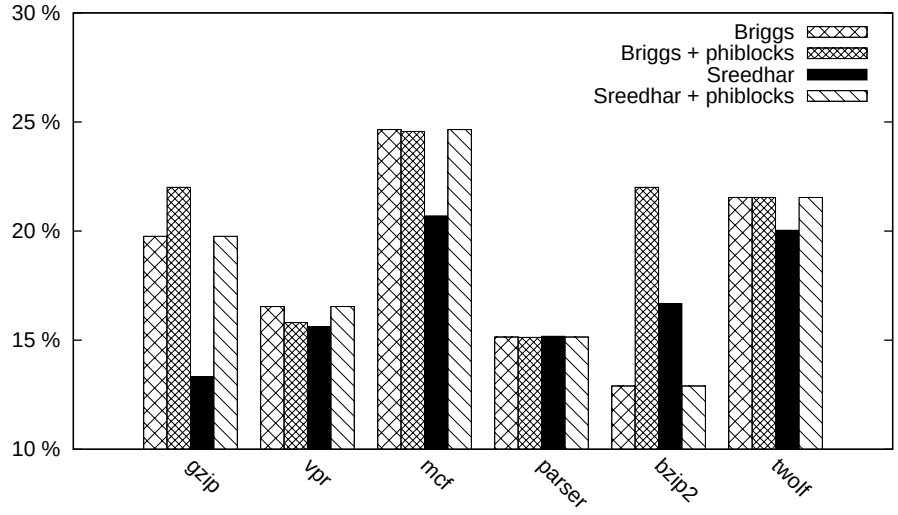
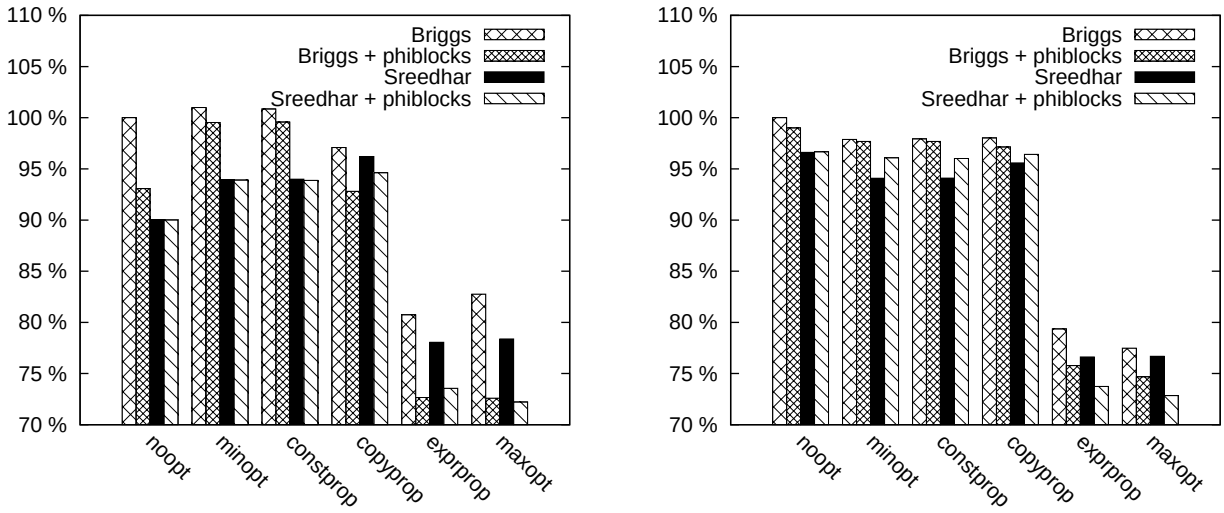


Figure 8.4: Improvement of post optimizations, expressed in percentage of the noopt execution time. Higher means more could be gained by post-optimizing, the end-result is not necessarily better.



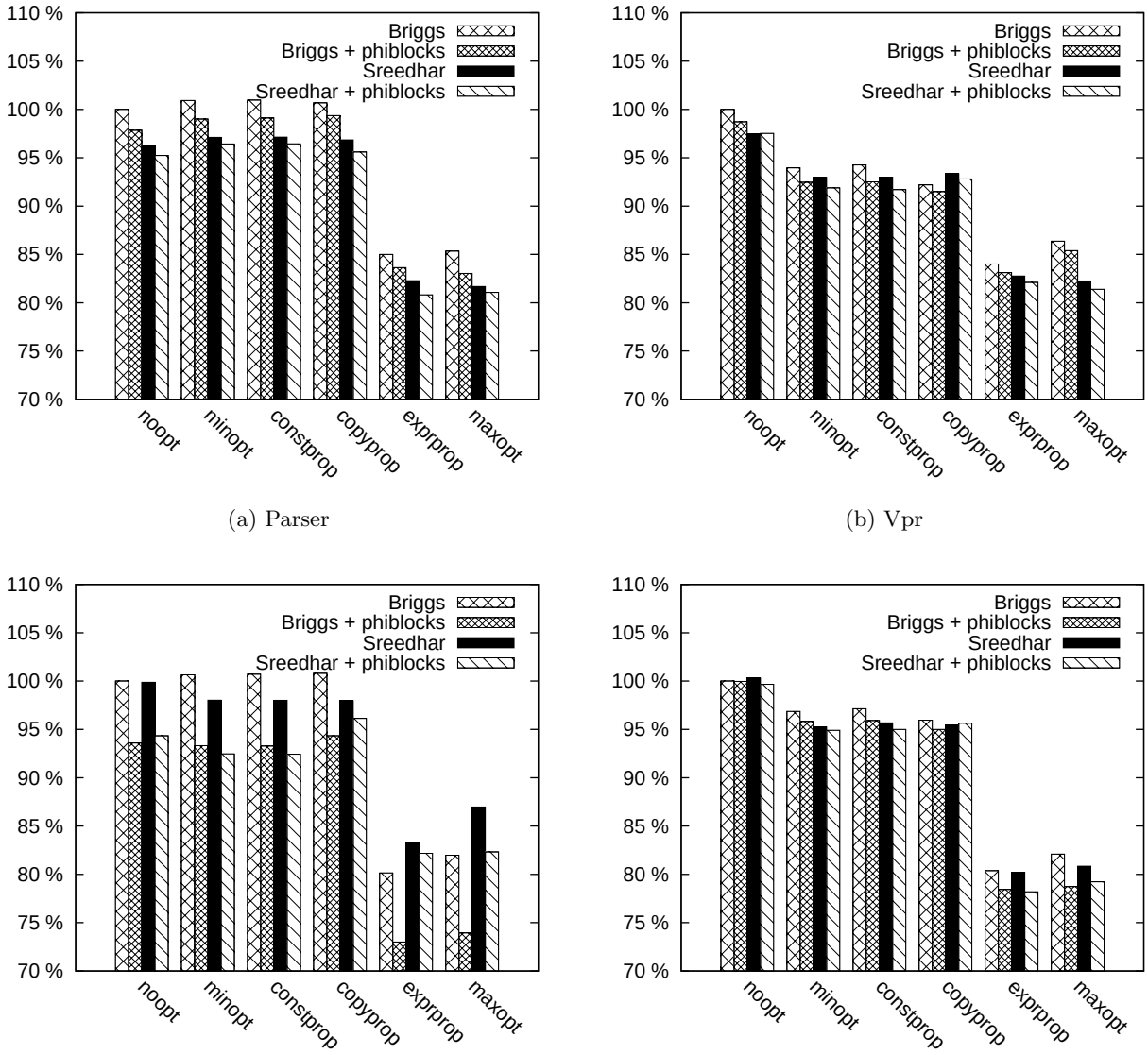
(a) Gzip

(b) Mcf

Figure 8.5: Ratios showing the effect of phiblock insertion and various post SSA back-translation optimizations on 2 of the benchmarks. All execution times are scaled to the time of the result when Briggs method is used for back-translation and no optimizations are applied afterwards. Also see Figure 8.6 in which the results of the other benchmarks are shown.

In some cases the *minopt* scheme worsens the result compared to applying no optimization at all. This may be the result of preparing the IR for copy, constant and expression propagation without actually performing the propagation.

Expression propagation is clearly the optimization pass with the greatest effect: in all cases the result after this pass is comparable to the result after applying all optimization.



(a) Parser

(b) Vpr

(c) Bzip2

(d) Twolf

Figure 8.6: Ratios showing the effect of phiblock insertion and various post SSA back-translation optimizations. (Cont. from Figure 8.5)

compiler options	gzip	vpr	mcf	parser	bzip2	twolf
briggs, noopt	31.995	1723.339	76.675	136.502	89.313	176.900
blocks with phiblocks, noopt	29.882	1701.251	75.584	133.468	84.475	176.949
sreedhar, noopt	29.248	1682.899	73.925	131.367	89.693	177.992
blocks with phiblocks, noopt	28.781	1680.741	74.170	129.924	84.633	176.604
briggs, minopt	32.293	1619.209	75.028	137.687	90.325	171.722
blocks with phiblocks, minopt	31.896	1593.288	74.655	135.063	83.477	169.168
sreedhar, minopt	30.243	1604.652	72.412	132.479	87.640	168.815
blocks with phiblocks, minopt	30.238	1586.074	73.743	131.738	83.035	168.120
briggs, constprop	32.258	1624.351	75.199	141.099	90.287	170.871
blocks with phiblocks, constprop	31.840	1594.278	74.804	135.212	83.553	168.666
sreedhar, constprop	30.239	1602.515	72.464	132.986	87.845	168.307
blocks with phiblocks, constprop	30.280	1584.467	73.668	131.763	83.100	167.878
briggs, copyprop	31.239	1588.906	75.013	137.346	90.959	170.515
blocks with phiblocks, copyprop	29.677	1576.702	74.823	135.548	86.708	167.168
sreedhar, copyprop	30.767	1628.212	73.239	132.130	87.611	167.768
blocks with phiblocks, copyprop	30.256	1614.013	73.800	130.421	87.535	168.588
briggs, exprprop	26.130	1447.844	61.498	116.241	71.840	142.977
blocks with phiblocks, exprprop	23.264	1432.357	58.157	114.093	66.190	139.425
sreedhar, exprprop	25.275	1426.318	58.645	112.249	74.461	143.048
blocks with phiblocks, exprprop	23.521	1414.849	56.426	110.253	73.537	138.073
briggs, maxopt	26.505	1488.478	59.165	116.418	73.335	146.379
blocks with phiblocks, maxopt	23.502	1471.699	57.467	114.315	66.698	138.673
sreedhar, maxopt	25.412	1417.429	58.690	111.429	77.792	142.961
blocks with phiblocks, maxopt	23.096	1402.712	55.723	111.169	73.610	141.012

Table 8.1: Execution times in seconds of all experiments.

Chapter 9

Conclusions

Compilers translate one representation of a software program into another. Internally a compiler uses an Intermediate Representation (IR) for analysis and manipulation of the program.

In this project two compilers are used. LLVM is an open-source compiler popular amongst academic research projects [11, 10]. CoSy is a compiler-framework build by ACE. For the research of this thesis a bridge was build that maps the IR of LLVM in SSA form onto CoSy in normal form. Various methods for the back-translation of SSA form exist, but the two main methods of back-translation are the methods of Briggs[3] and Sreedhar[13]. The main isssue is how to translate phi functions, which merge control flow streams, to normal form.

Briggs' method emits copy instructions for all variables involved in the phi function. Sreedhar's method emits fewer copy instructions and reduces the number of variables by coalescing. Since optimization is performed before before back-translation, abundant use of copy instructions may have a negative impact on execution time. Sassa showed that Sreedhar gives a faster result[12].

A novel modification is proposed to split edges by inserting phiblocks for the methods of Sreedhar en Briggs. The modification is tested by implementing the original and modified methods of Sreedhar and Brigs and use them in the LLVM-CoSy bridge.
This allowed for answering the following questions:

- Can the back-translation methods of Briggs and Sreedhar undo optimizations that are performed before going out of SSA? (Yes.)
- What is the effect of placing instructions in phiblocks compared to the conventional methods of Sreedhar and Briggs on the execution time of the resulting code? (Faster results.)

- Previous research of Sassa showed that Sreedhar's method is superior to Briggs' method. Is this also true when the phiblock modification is applied? (Incidentally Briggs' result becomes as good as Sreedhar's, on average Sreedhar's results are still better.)

The benchmarks showed that *the resulting IR of back-translating an optimized IR in SSA form with the methods of Briggs and Sreedhar is sub-optimal.* This is shown by the fact that optimizations such as propagation after back-translation speed up the result on average with 18% even though the same optimization is already done in-SSA.

The swap problem is used to show why edge splitting with phiblocks can be beneficial. The benchmarks results show that *the phiblock modification of Briggs and Sreedhar speeds up results that are already optimized before and after back-translation*: execution time is reduced on average by 5% for Brigg's method and 3% for Sreedhar's method. These are improvements over a result that is already optimized with LLVM and with the set of post-optimizations in CoSy.

Briggs' algorithm is much simpler and less prone to mistakes in the implementation than the algorithm of Sreedhar. When phiblocks and post optimization are used, on average the results of Sreedhar's method are faster by 2% compared to Briggs' result. For certain cases, however, *by using phiblocks and post optimization the difference between the results of the simple back-translation method of Briggs are almost as good as the result of Sreedhar's method.* In case of the SPEC gzip benchmark: when no post optimization or phiblocks are used Sreedhar's result is 10% faster than Briggs's result but by using post optimization and phiblock insertion the difference is reduced to less than 1%.

Left for future research on this topic is the effect of phiblocks when more limited architectures are targeted. Another interesting question is if a set of optimization techniques exists that can nullify the difference between Sreedhar and Briggs with an acceptable increase in compilation time.

Bibliography

[1] F.E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[2] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting out-of-ssa translation for correctness, code quality and efficiency. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28:859–881, July 1998.

[4] D. Budgen. *Software design*. Addison Wesley, 2003.

[5] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *In Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation (PLDI-02*, pages 25–32. ACM Press, 2002.

[6] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17:98–101, June 1982.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[8] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in ssa-form. In *In Compiler Construction 2006, volume 3923 of LNCS*, pages 247–262. Springer Verlag, 2006.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007.

[10] List of LLVM related publications. http://llvm.org/pubs/, Feb. 2011.

[11] List of LLVM users. http://llvm.org/users.html, Dec. 2010.

[12] Masataka Sassa and Masaki Kohama. Comparison and evaluation of back-translation algorithms for static single assignment forms. *Comput. Lang. Syst. Struct.*, pages 173–195, July 2009.

[13] Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *In Static Analysis Symposium, Venezia, Italy*, pages 194–210. Springer Verlag, 1999.

Index