Automatic Validation of Code-Improving Transformations on Low-Level Program Representations *

Robert van Engelen, David Whalley, and Xin Yuan
Department of Computer Science, Florida State University, Tallahassee, FL 32306

Abstract

This paper presents a general approach to automatically validate code-improving transformations on low-level program representations. The approach ensures the correctness of compiler and hand-specified optimizations at the machine instruction level. The method verifies the semantic equivalence of the program representation before and after a transformation to determine the validity of the transformation, i.e. whether the instance of the transformation is semantics preserving. To verify that the transformation is semantics preserving, the method derives semantic effects from the sequence of low-level instructions that span the execution paths affected by the transformation. The semantics are preserved if the normalized semantic effects derived from the code before and after the transformation are proven to be identical. A validating compilation system was implemented that is able to validate basic changes comprising instruction insertions, modifications, and deletions, to more powerful transformations that modify the branch structure of a function in a program.

1 Introduction

Software is being used as a component of an increasing number of critical systems. Ensuring that these systems execute correctly is vital. Because software is incorporated in an increasing number of critical systems, there is a need to ensure that compilers produce machine code that correctly represents the algorithms specified at the source code level. This is a formidable task since an optimizing compiler translates a source code program to machine code while applying hundreds or thousands of compiler optimizations to even a relatively small program.

To ensure that a compiler produces correct machine code, compiler developers must guarantee that all compiler optimizations are semantics preserving. Proving the correctness of optimizing transformations is a challenging problem in general [8, 11, 12, 16, 17, 20, 21, 23, 24]. Compiler optimizations are often quite complex in terms of the computational power of the analysis algorithms used for detecting opportunities for optimizations and the intricacy of the code transformations applied to exploit specific architectural features of a target machine. The problem of proving the semantics preserving property of optimizing transformations is exacerbated for embedded systems development [26], where often either applications are developed in assembly code manually or compiler-generated assembly is modified by hand to meet speed and/or space constraints. This is an important problem for embedded system developers, because the cost of malfunctioning software in embedded systems is huge. For example, single chip designs featuring on-chip program code do not support the installation of software updates when an error is discovered after production.

*This work was partially supported by NSF grants, CCR-9904943, EIA-0072043, CCR-0073482, CCR-0105422, CCR-0208892, and by DOE grant DEFG02-02ER25543

There has been much work done in the area of attempting to prove the correctness of compilers [11, 12, 16, 17, 23]. However, proving the correctness of production-quality compilers that apply complex and intricate optimizations has been found to be difficult or impossible in many cases. More success has been made in the area of validating compilations rather than proving the correctness of the compiler itself [8, 20, 21, 24]. Therefore, it is perceived that proving the semantics preserving property of the instance of a transformation is more practical.

This paper follows the latter approach and presents a general method to automatically validate code-improving optimizations on low-level program representations by verifying the correctness of the compilation process at the machine instruction level. This ensures the correctness of compiler optimizations and hand-specified modifications of the code. The method verifies the semantic equivalence of the program representation before and after a transformation to determine the validity of the transformation, i.e. whether the rule instance of the transformation is semantics preserving. Each transformation may consist of a fixed sequence of simple atomic modifications to the code, such as instruction insertions and deletions, to achieve one optimization. The intermediate stages of such a sequence of atomic modifications may not be required to be semantics preserving, but it is assumed that the end result of the sequence of modifications, constituting one optimizing transformation, preserves the semantics of the code. To verify the semantics preserving property of a transformation, the method derives a set of semantic effects from the sequence of low-level instructions that span the execution paths affected by the transformation. The semantics are preserved if the semantic effects of the code before and after each transformation are identical. A validating compilation system was implemented that is able to validate basic changes comprising instruction insertions, modifications, and deletions, to more powerful transformations that modify the branch structure of a function in a program.

This paper is an updated and extended version of two earlier publications. The idea to validate optimizing transformations using semantic effects was first introduced in [24]. A second paper [26] describes the applicability of the approach to validate the optimization of embedded software using an interactive compilation system for code development.

The remainder of this paper is organized as follows. An overview of compiler validation methods and related work is presented in Section 2. In Section 3 the register transfer list notation is introduced for representing low-level machine instructions. Section 4 presents an introduction to modeling semantics of low-level instructions using register transfer lists, where the model is based on the strongest postcondition from Hoare logic. Section 5 describes the algorithms and implementation of the automatic compiler validation system and results on a number of benchmark programs are presented in Section 6. The paper is summarized with some concluding remarks in Section 7.

2 An Overview of Compiler Validation Methods and Related Work

Previous work on compiler validation attempted to prove the correctness of compilers [11, 12, 16, 17, 23]. However, more success has been made in the area of validating compilations [8, 20, 21, 24].

The *credible compilation* approach [21] attempts to validate code-improving transformations on an intermediate machine-independent representation, but uses a different approach from the one described in this paper. The compiler writer determines the appropriate type of *invariants* for the analysis and transformation of each different type of code-improving optimization and the compiler automatically constructs a proof for these invariants for each optimization. While this approach is quite powerful, it puts a burden on a compiler writer to correctly identify the types of changes associated with each optimization and to specify the appropriate invariants that need to be proven correct. In addition, the invariants are assumed to be cor-

rect. Otherwise, incorrect optimizations may pass a proof of correctness based on incorrect invariants (false positives) and correct optimizations may be rejected (false negatives).

The work most closely related to ours is by Necula [20]. The method calculates a symbolic state of each *basic block* [1] and then uses equivalence relations to prove that two blocks are equivalent when the symbolic states of blocks before and after a transformation are identical. His work demonstrates that most optimization phases can be validated during the compilation of the *gcc* source code by the *gcc* compiler itself. However, the validation system is not able to validate transformations that change the *branch structure* of the *control flow graph* (CFG) [1], which is caused by transformations that optimize the *control flow* of the code and other optimizations that span multiple basic blocks in the CFG. This work differs from ours in that it is more restrictive. The approach presented in this paper can validate transformations that change the branch structure of the CFG.

Closely related is the work on proving *semantic equivalence* of textually different source programs. Bergstra [5] developed a method based on normal forms to prove the semantic equivalence of textually different source programs. However, this has only been attempted on a restricted high-level language without loops and function calls. Horwitz [14] attempted to identify semantic differences between source programs in a simple high-level language containing a limited number of constructs. First, programs in this language were represented in a *program representation graph* that identifies dependences between statements. Next, a matching function examined both representations to determine if *old* and *new* representations were equivalent. A similar approach using a matching function could be applied on a low-level representation to validate some types of transformations, such as those that change the *order* of independent instructions. However, it is unclear how other transformations, such as the those that change the *form* of instructions, could be validated using Horowitz' approach.

Capturing the changes associated with compiler transformations has been used by a number of related projects that were concerned with debugging either the compiler or the optimized code it produced. For example, compiler visualization systems have been developed to illustrate the differences in the program representation before and after a transformation [7, 10]. Mappings between unoptimized and optimized programs have been produced to facilitate debugging of optimized code [15].

There has also been work in isolating the code-improving transformation that causes an error in the code when a runtime error has been discovered by testing the optimized code on an input data set [6, 29]. Unfortunately, it is usually impractical to provide complete test coverage of compiled programs. Therefore, validation of the compiler and the transformation it applied is preferable over isolating a transformation that caused a runtime error.

3 Register Transfer Lists

The *register transfer list* (RTL) notation is a popular intermediate code representation for low-level instructions used by a variety of compilers, such as *gcc* and *vpo* [4]. The RTL notation is uniform and provides an orthogonal instruction set based on predicated assignments. The RTL notation used by *gcc* is distinctly more Lisp-like compared to the vpo RTL. However, the two RTL variants are conceptually the same. Other intermediate representations used by compilers, such as three-address code and triples [1] for example, can be easily mapped to RTL instructions. Because most compilers adopt RTL or a representation that is isomorphic to RTL, the validation of the transformations at the RTL level presented in this paper provides a generic approach.

The Verilog register transfer level model (Verilog RTL) is somewhat related to register transfer lists. However, Verilog RTL provides a more comprehensive instruction set for the verifiable synthesis and simu-

RTL	$\rightarrow$	L = E ;	assignment
	|	L = E_1 , E_2 ;	predicated assignment (assigns E_2 to L conditional on E_1)
	|	*RTL* \n *RTL*	sequential execution (\n indicates newline)
	|	*RTL RTL*	concurrent execution (RTLs on a single line)
E	$\rightarrow$	$\ominus E$	monadic operation
	|	$E \oplus E$	dyadic operation
	|	*label*	address, offset, or basic block label
	|	*intConst*	integer constant
	|	L	value of an l-value
L	$\rightarrow$	r[$natConst$]	indexed (virtual) register
	|	M[E]	memory reference
	|	PC	program counter
	|	IC	condition code register
	|	RT	return address (to caller)
$\ominus$	$\rightarrow$	$- \mid \neg$	negation and binary complement
$\oplus$	$\rightarrow$	$\wedge \mid \bar{\wedge} \mid \vee \mid \bar{\vee} \mid \uplus$	and, nand, or, nor, xor logical operators
	|	$< \mid \leq \mid > \mid \geq \mid = \mid \neq$	relational operators
	|	?	compare operator (see text)
	|	$+ \mid -$	addition and subtraction
	|	$* \mid / \mid$ %	multiplication, division, and modulo
	|	$\ll \mid \gg$	binary arithmetic shift

Figure 1: RTL Grammar

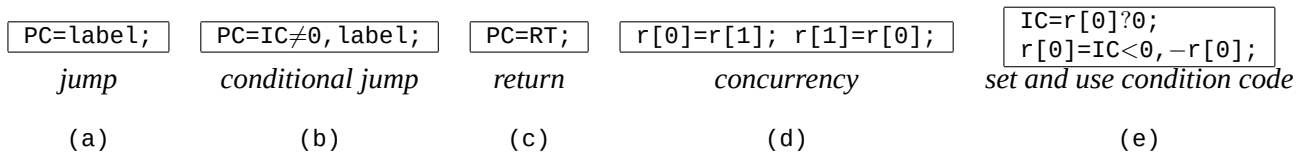


Figure 2: Examples

lation of electronic systems [3].

The *vpo* compiler applies code-improving transformations at the RTL level. After optimization, the RTL representation of a program is converted to machine code by a post-processing stage in the back-end of the compiler. The RTL notation used by *vpo* is expressed with the grammar[1] shown in Figure 1. The RTL language is composed of assignments to registers and memory locations. Conditional and unconditional control flow is specified with predicated assignments to the program counter PC. Figure 2(a), (b), and (c) illustrate examples of an unconditional jump instruction, a conditional jump instruction, and a return statement, respectively. These instructions are used to build the *control flow graph* structure [1]. A CFG of a source-code function is composed of nodes that are called *basic blocks*, where each basic block contains a sequence of consecutive RTL instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [1].

RTL instructions can be concurrently executed as is illustrated in Figure 2(d). The concurrent execution of RTLs respects copy-in copy-out semantics. Thus, the example shown in the figure exchanges the values of the r[0] and r[1] registers. Concurrent RTL instructions can be used to describe multiple parallel data transfers in hardware, such as auto-increment registers that access memory and update the address they are

[1]The grammar shown in Figure 1 is ambiguous. It is assumed that the usual associative and precedence properties of the arithmetic operators and parenthesis are used for disambiguation.

pointing to during the execution of the memory load or store operation.

The compare operator is used to compare two values, basically using subtraction with infinite precision. The actual result of the compare operator is discarded, only the condition code flags are set. Therefore, this operator is only relevant for RTL instructions that set the condition code register `IC`, as is illustrated in Figure 2(e). In this example, the first RTL instruction sets the condition code register `IC` by comparing register `r[0]` to 0. The second (predicated) instruction negates `r[0]` if the `IC` sign flag is negative.

Floating point numbers can be used with RTL programs, but not directly. Instead, floating point values are loaded from a static memory region or registers are set using the (multiple) integer values of floating point standard representations, such as IEEE 754. This is important, because the modeling of low-level semantics is simplified when restricted to integer-based arithmetic and floating point operations that may possibly cause roundoff errors can be avoided.

4 Modeling Low-Level Program Semantics

This section introduces the modeling of low-level program code semantics using the strongest postcondition rule of axiomatic semantics. It is shown that the computation of the semantic effects of a region of code is identical to the derivation of the strongest postcondition for that region.

4.1 Strongest Postcondition

RTL instructions consist of assignments to registers and memory. Hoare logic [13] is used to model the semantics of RTL instructions. The Hoare triple for the strongest postcondition for assignment [9] is defined by

$$\{P\} \quad L\texttt{=}E\texttt{;} \quad \{\, \exists \ell . P[\ell/L] \wedge L = E[\ell/L] \,\}$$

where $\exists \ell . P[\ell/L] \wedge L = E[\ell/L]$ is the postcondition derived from the precondition P, with $[\ell/L]$ denoting the substitution of L by the term ℓ as defined in Figure 3. The term L is any l-value except `PC`, because RTL assignments that influence the flow of control must be handled differently. The semantic modeling of RTL instructions that influence the flow of control will be discussed in Section 5. The Hoare triple for the RTL predicated assignment is defined by

$$\{P\} \quad L\texttt{=}E_1\texttt{,}E_2\texttt{;} \quad \{\, \exists \ell . P[\ell/L] \wedge L = \left\{ \begin{array}{ll} E_2 & if\ E_1 \\ L & if\ \neg E_1 \end{array} \right\} [\ell/L] \,\}$$

where L is any l-value except `PC`. Note that the rule for the RTL predicated assignment requires conditioning the r-value E_2 on E_1, where E_2 is assigned to L if E_1 is true and L is assigned to itself otherwise. The axiomatic rules for concurrent (predicated) assignments are similar and are based on the concept of simultaneous substitutions.

The substitution algorithm shown in Figure 3 is an extended substitution algorithm to model potential aliased pointers introduced by the memory reference construct `M[`E`]` in the RTL language, where the value of the address expression E may not be known at compile time. Registers cannot be aliased, thus the presence of aliases is limited to memory references. The substitution algorithm replaces a register or a memory reference with a replacement term in an RTL expression. The RTL expression may contain an additional conditional construct, denoted with curly braces, introduced for representing predicated instructions, conditional control flow, and potential aliases in the semantic model.

The application of the (predicated) assignment rule to a sequence of RTL instructions is straight forward. It is assumed that the initial precondition P_0 models the entire input state of the machine as defined by

$$
\begin{array}{rcll}
(\ominus E_1)[E/\mathtt{r}[n]] & \mapsto & \ominus E_1[E/\mathtt{r}[n]] & \\
(E_1 \oplus E_2)[E/\mathtt{r}[n]] & \mapsto & E_1[E/\mathtt{r}[n]] \oplus E_2[E/\mathtt{r}[n]] & \\
\mathit{label}[E/\mathtt{r}[n]] & \mapsto & \mathit{label} & \\
\mathit{intConst}[E/\mathtt{r}[n]] & \mapsto & \mathit{intConst} & \\
(\mathtt{r}[k])[E/\mathtt{r}[n]] & \mapsto & \mathtt{r}[k] & (\text{when } n \neq k) \\
(\mathtt{r}[n])[E/\mathtt{r}[n]] & \mapsto & E & \\
(\mathtt{M}[E_1])[E/\mathtt{r}[n]] & \mapsto & \mathtt{M}[E_1[E/\mathtt{r}[n]]] & \\
\left\{ \begin{array}{ll} E_1 & if\ E_{k+1} \\ \vdots & \\ E_k & if\ E_{2k} \end{array} \right\}[E/\mathtt{r}[n]] & \mapsto & \left\{ \begin{array}{ll} E_1[E/\mathtt{r}[n]] & if\ E_{k+1}[E/\mathtt{r}[n]] \\ \vdots & \\ E_k[E/\mathtt{r}[n]] & if\ E_{2k}[E/\mathtt{r}[n]] \end{array} \right\} & \\
(\ominus E_1)[E/\mathtt{M}[E']] & \mapsto & \ominus E_1[E/\mathtt{M}[E']] & \\
(E_1 \oplus E_2)[E/\mathtt{M}[E']] & \mapsto & E_1[E/\mathtt{M}[E']] \oplus E_2[E/\mathtt{M}[E']] & \\
\mathit{label}[E/\mathtt{M}[E']] & \mapsto & \mathit{label} & \\
\mathit{intConst}[E/\mathtt{M}[E']] & \mapsto & \mathit{intConst} & \\
(\mathtt{r}[n])[E/\mathtt{M}[E']] & \mapsto & \mathtt{r}[n] & \\
(\mathtt{M}[E_1])[E/\mathtt{M}[E']] & \mapsto & \left\{ \begin{array}{ll} E & if\ E' = E_1 \\ \mathtt{M}[E_1] & if\ E' \neq E_1 \end{array} \right\} & \\
\left\{ \begin{array}{ll} E_1 & if\ E_{k+1} \\ \vdots & \\ E_k & if\ E_{2k} \end{array} \right\}[E/\mathtt{M}[E']] & \mapsto & \left\{ \begin{array}{ll} E_1[E/\mathtt{M}[E']] & if\ E_{k+1}[E/\mathtt{M}[E']] \\ \vdots & \\ E_k[E/\mathtt{M}[E']] & if\ E_{2k}[E/\mathtt{M}[E']] \end{array} \right\} &
\end{array}
$$

Figure 3: Substitution

$$P_0 = (\textstyle\bigwedge_{n=0}^{N} \mathtt{r}[n] = \mathtt{r}_0[n]) \wedge (\bigwedge_{m=0}^{M} \mathtt{M}[m] = \mathtt{M}_0[m])$$

Thus, for defining the precondition each register $\mathtt{r}[n]$, $n = 0, \ldots, N$, is bound to an initial value $\mathtt{r}_0[n]$ and each of the $m = 0, \ldots, M$ memory locations $\mathtt{M}[m]$ holds an initial value $\mathtt{M}_0[m]$. With this assumption, the choice of value ℓ in the postcondition of an RTL assignment instruction is simply the current value (state) of the l-value L given in the precondition of the assignment instruction. An example representative sequence of RTL instructions is given below and is intended to illustrate the application of the axiomatic semantic rules:

$\{P_0\}$
r[0]=r[0]+1;
$\{P_1 \wedge$ r[0]=$\mathtt{r}_0$[0]+1 $\}$
r[1]=r[0]-1;
$\{P_2 \wedge$ r[0]=$\mathtt{r}_0$[0]+1 $\wedge$ r[1]=r[0] $- 1$ $\}$
M[r[1]]=M[r[2]];
$\{P_3 \wedge$ r[0]=$\mathtt{r}_0$[0]+1 $\wedge$ r[1]=r[0] $- 1 \wedge$ M[r[1]]$= \left\{ \begin{array}{ll} \mathtt{M}_0[\mathtt{r}[1]] & if\ \mathtt{r}[1] = \mathtt{r}[2] \\ \mathtt{M}[\mathtt{r}[2]] & if\ \mathtt{r}[1] \neq \mathtt{r}[2] \end{array} \right\}$ $\}$
r[0]=M[r[1]];
$\{P_4 \wedge$ r[0]=M[r[1]] $\wedge$ r[1]=$\mathtt{r}_0$[0] $+ 1 - 1 \wedge$ M[r[1]]$= \left\{ \begin{array}{ll} \mathtt{M}_0[\mathtt{r}[1]] & if\ \mathtt{r}[1] = \mathtt{r}[2] \\ \mathtt{M}[\mathtt{r}[2]] & if\ \mathtt{r}[1] \neq \mathtt{r}[2] \end{array} \right\}$ $\}$

where

$$
\begin{array}{lcl}
P_1 & = & P_0[\mathtt{r}_0\mathtt{[0]}/\mathtt{r[0]}] = (\bigwedge_{n=1}^{N} \mathtt{r}[n] = \mathtt{r}_0[n]) \wedge (\bigwedge_{m=0}^{M} \mathtt{M}[m] = \mathtt{M}_0[m]) \\
P_2 & = & P_1[\mathtt{r}_0\mathtt{[1]}/\mathtt{r[1]}] = (\bigwedge_{n=2}^{N} \mathtt{r}[n] = \mathtt{r}_0[n]) \wedge (\bigwedge_{m=0}^{M} \mathtt{M}[m] = \mathtt{M}_0[m]) \\
P_3 & = & P_2[\mathtt{M}_0\mathtt{[r[1]]}/\mathtt{M[r[1]]}] \\
 & = & (\bigwedge_{n=2}^{N} \mathtt{r}[n] = \mathtt{r}_0[n]) \wedge (\bigwedge_{m=0}^{M} \left\{ \begin{array}{ll} \mathtt{M}_0\mathtt{[r[1]]} & if\ \mathtt{r}[1] = m \\ \mathtt{M}[m] & if\ \mathtt{r}[1] \neq m \end{array} \right\} = \mathtt{M}_0[m]) \\
P_4 & = & P_3[(\mathtt{r}_0\mathtt{[0]}+1)/\mathtt{r[0]}] \\
 & = & (\bigwedge_{n=2}^{N} \mathtt{r}[n] = \mathtt{r}_0[n]) \wedge (\bigwedge_{m=0}^{M} \left\{ \begin{array}{ll} \mathtt{M}_0\mathtt{[r[1]]} & if\ \mathtt{r}[1] = m \\ \mathtt{M}[m] & if\ \mathtt{r}[1] \neq m \end{array} \right\} = \mathtt{M}_0[m])
\end{array}
$$

The rules are applied from the top to the bottom and the non-normalized intermediate conditions are shown. Normalization of the final postcondition leads to

$$\{P_4 \ \wedge\ \mathtt{r[0]}{=}\mathtt{M}_0\mathtt{[r}_0\mathtt{[2]]} \ \wedge\ \mathtt{r[1]}{=}\mathtt{r}_0\mathtt{[0]} \ \wedge\ \mathtt{M[r}_0\mathtt{[0]]}{=}\mathtt{M}_0\mathtt{[r}_0\mathtt{[2]]} \ \}$$

where P_4 is normalized to

$$
\begin{array}{lcl}
P_4 & = & (\bigwedge_{n=2}^{N} \mathtt{r}[n] = \mathtt{r}_0[n]) \wedge (\bigwedge_{m=0}^{M} \left\{ \begin{array}{ll} \mathtt{M}_0\mathtt{[r}_0\mathtt{[0]]} & if\ \mathtt{r}_0[0] = m \\ \mathtt{M}[m] & if\ \mathtt{r}_0[0] \neq m \end{array} \right\} = \mathtt{M}_0[n]) \\
 & = & (\bigwedge_{n=2}^{N} \mathtt{r}[n] = \mathtt{r}_0[n]) \wedge (\bigwedge_{m=0,(m\neq \mathtt{r}_0[0])}^{M} \mathtt{M}[m] = \mathtt{M}_0[m])
\end{array}
$$

The normalization process replaces all occurrences of registers and memory references on the equation's right-hand sides with their r-values using the substitution algorithm. Arithmetic and logical simplification are applied to reduce the resulting expressions. A subset of the rules for expression simplification is shown in Figure 4.

4.2 RTL Effects

The semantics of a region of code are modeled using a simplified form of postconditions, called RTL *effects* [24]. Effects describe the impact of the instructions on registers and memory. This simplified form has the same meaning as a postcondition in axiomatic semantics, but uses a different notation. The mapping of postconditions to effects is defined as follows:

- The P_0, P_1, P_2, ..., are eliminated from the postconditions to obtain effects. These can be regenerated at any time by inspecting the postcondition's free variables, which are the registers and memory references that make up the equations in the main part of the postcondition, i.e. the postcondition without the P_i part.

- Effects are always normalized (via substitution and simplification), therefore all registers and memory references that were set in the region of code occur on the left-hand sides of the equations. The registers and memory references occurring on the right-hand sides are always initial values of registers and memory, i.e. $\mathtt{r}_0\mathtt{[}n\mathtt{]}$ and $\mathtt{M}_0\mathtt{[}m\mathtt{]}$, respectively.

- Equations for registers that are dead are eliminated from effects. The compiler annotates RTL instructions with register liveness information based on *def-use data flow analysis* [1]. When a register or a memory reference (i.e. a program variable) is flagged as dead at a certain point in the code, its equation describing the effect on the register is removed. Registers provide temporary storage and are therefore not critical to the modeling of semantics of a program, unlike memory updates.

$(E_1 \vee E_2) \wedge E_3$	$\Rightarrow$	$(E_1 \wedge E_3) \vee (E_2 \wedge E_3)$	distribution
$\neg(E_1 \vee E_2)$	$\Rightarrow$	$\neg E_1 \wedge \neg E_2$	De Morgan
$\neg(E_1 \wedge E_2)$	$\Rightarrow$	$\neg E_1 \vee \neg E_2$	De Morgan
$E_1 \vee (E_1 \wedge E_2)$	$\Rightarrow$	E_1	absorption
$E_1 \vee (\neg E_1 \wedge E_2)$	$\Rightarrow$	$E_1 \vee E_2$	absorption
$\neg E_1 \vee (E_1 \wedge E_2)$	$\Rightarrow$	$\neg E_1 \vee E_2$	absorption
$(E_1 \wedge E_2) \vee (\neg E_1 \wedge E_2)$	$\Rightarrow$	E_2	absorption
$(E_1 \wedge E_2 \wedge E_3) \vee (\neg E_1 \wedge E_3)$	$\Rightarrow$	$(E_2 \wedge E_3) \vee (\neg E_1 \wedge E_3)$	absorption
$(\neg E_1 \wedge E_2 \wedge E_3) \vee (E_1 \wedge E_3)$	$\Rightarrow$	$(E_2 \wedge E_3) \vee (E_1 \wedge E_3)$	absorption
$\left\{ \begin{array}{ll} E_1 & if\ E_2 \\ \vdots & if \vdots \end{array} \right\} if\ E_3$	$\Rightarrow$	$\left\{ \begin{array}{ll} E_1 & if\ E_2 \wedge\ E_3 \\ \vdots & if \vdots \wedge E_3 \end{array} \right\}$	left un-nesting
$E_1\ if \left\{ \begin{array}{ll} E_2 & if\ E_3 \\ \vdots & if \vdots \end{array} \right\}$	$\Rightarrow$	$\left\{ \begin{array}{ll} E_1 & if\ E_2 \wedge E_3 \\ E_1 & if \vdots \wedge \vdots \end{array} \right\}$	right un-nesting
$\left\{ \begin{array}{l} \dots \\ E_1 \quad if\ E_2 \\ \dots \\ E_1 \quad if\ E_3 \\ \dots \end{array} \right\}$	$\Rightarrow$	$\left\{ \begin{array}{l} \dots \\ E_1 \quad if\ E_2 \vee E_3 \\ \dots \end{array} \right\}$	combining arms
$E_1 \oplus \left\{ \begin{array}{ll} E_2 & if\ E_3 \\ \vdots & if \vdots \end{array} \right\}$	$\Rightarrow$	$\left\{ \begin{array}{ll} E_1 \oplus E_2 & if\ E_3 \\ E_1 \oplus \vdots & if \vdots \end{array} \right\}$	promoting
$\left\{ \begin{array}{ll} E_1 & if\ E_2 \\ \vdots & if \vdots \end{array} \right\} \oplus E_3$	$\Rightarrow$	$\left\{ \begin{array}{ll} E_1 \oplus E_3 & if\ E_2 \\ \vdots \oplus E_3 & if \vdots \end{array} \right\}$	promoting
$\ominus \left\{ \begin{array}{ll} E_2 & if\ E_3 \\ \vdots & if \vdots \end{array} \right\}$	$\Rightarrow$	$\left\{ \begin{array}{ll} \ominus E_2 & if\ E_3 \\ \ominus \vdots & if \vdots \end{array} \right\}$	promoting
$E_1 = E_2$	$\Rightarrow$	$E_1 - E_2 = 0$	when $E_1 - E_2 <_{\text{lex}} E_2 - E_1$ and $E_2 \neq 0$
$E_1 = E_2$	$\Rightarrow$	$E_2 - E_1 = 0$	when $E_1 - E_2 \geq_{\text{lex}} E_2 - E_1$ and $E_2 \neq 0$
$E_1 > E_2$	$\Rightarrow$	$E_1 - E_2 > 0$	when $E_1 - E_2 <_{\text{lex}} E_2 - E_1$ and $E_2 \neq 0$
$E_1 > E_2$	$\Rightarrow$	$\neg(E_2 - E_1 + 1 > 0) \vee \neg(E_2 - E_1 > 0)$	when $E_1 - E_2 \geq_{\text{lex}} E_2 - E_1$ and $E_2 \neq 0$
$E_1 \geq E_2$	$\Rightarrow$	$E_1 - E_2 + 1 > 0$	when $E_1 - E_2 <_{\text{lex}} E_2 - E_1$ and $E_2 \neq 0$
$E_1 \geq E_2$	$\Rightarrow$	$\neg(E_2 - E_1 > 0)$	when $E_1 - E_2 \geq_{\text{lex}} E_2 - E_1$ and $E_2 \neq 0$
$E_1 \neq E_2$	$\Rightarrow$	$\neg(E_1 - E_2 = 0)$	when $E_1 - E_2 <_{\text{lex}} E_2 - E_1$ and $E_2 \neq 0$
$E_1 \neq E_2$	$\Rightarrow$	$\neg(E_2 - E_1 = 0)$	when $E_1 - E_2 \geq_{\text{lex}} E_2 - E_1$ and $E_2 \neq 0$
$E_1 > 0 \wedge E_2 > 0$	$\Rightarrow$	$E_1 > 0$	when $E_1 \leq E_2$
$E_1 > 0 \wedge E_2 > 0$	$\Rightarrow$	$E_2 > 0$	when $E_1 > E_2$
$E_1 > 0 \wedge E_2 = 0$	$\Rightarrow$	$false$	when $E_1 \leq E_2$
$E_1 > 0 \wedge E_2 = 0$	$\Rightarrow$	$E_2 = 0$	when $E_1 > E_2$
$E_1 > 0 \vee E_2 > 0$	$\Rightarrow$	$E_2 > 0$	when $E_1 \leq E_2$
$E_1 > 0 \vee E_2 > 0$	$\Rightarrow$	$E_1 > 0$	when $E_1 > E_2$
$E_1 > 0 \vee E_2 = 0$	$\Rightarrow$	$E_1 + 1 > 0$	when $E_1 = E_2$
$E_1 > 0 \vee E_2 = 0$	$\Rightarrow$	$E_1 > 0$	when $E_1 > E_2$

Figure 4: Simplification

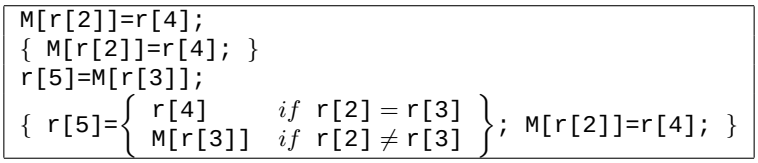


Figure 5: Potential Def-Use Alias

The simplified effect forms enable a faster derivation of the semantic description by a process called *merging* [24]. Merging effects requires substitution and normalization, similar to deriving postconditions.

Consider for example the following region of code annotated with effects derived by merging, i.e. through substitution and normalization:

```
r[0]=r[0]+1;
{ r[0]=r0[0]+1 }
r[1]=r[0]-1;                                              r[0]:
{ r[1]=r0[0] }
M[r[1]]=M[r[2]];
{ r[1]=r0[0] ∧ M[r0[0]]=M0[r0[2]] }
r[0]=M[r[1]];
{ r[0]=M0[r0[2]] ∧ r[1]=r0[0] ∧ M[r0[0]]=M0[r0[2]] }
```

In each step, the effects of the previous RTL instructions are merged with the current RTL effect using substitution and normalization. Note that the annotation `r[0]:` indicates that register `r[0]` is not live in the region of code below the annotated instruction. That is, the instruction is the last to use the live value of `r[0]` set by the first instruction. Therefore, the RTL effect on `r[0]` was eliminated in the second step.

It is guaranteed that effects produced by the application of the substitution algorithm in a manner identical to the application of substitution in the postcondition rule for assignments, consist of equations whose left-hand sides are l-values, i.e. registers and memory references. As a result, effects can be viewed as concurrent RTL constructs themselves, such as

$$\texttt{r[0]=M}_0\texttt{[r}_0\texttt{[2]]; r[1]=r}_0\texttt{[0]; M[r}_0\texttt{[0]]=M}_0\texttt{[r}_0\texttt{[2]];}$$

These RTL effects are state modifiers, providing a means to directly describe the impact of the original sequence of instructions on the registers and memory of a machine.

Because effects can be written as concurrent RTL assignments that obey copy-in copy-out semantics, the notations $\texttt{r}_0\texttt{[}n\texttt{]}$ and $\texttt{M}_0\texttt{[}m\texttt{]}$ have become redundant as a means to express initial values. In the sequel, these notations will be dropped all together in favor of a further compressed notation, such as

```
r[0]=M[r[2]]; r[1]=r[0]; M[r[0]]=M[r[2]];
```

for the example shown above.

4.3 Guarded RTL Effects

Guarded expressions in RTL effects express control flow, predication, and potential aliasing. With respect to aliasing, consider the merging of effects for the region of code shown in Figure 5, where a *def* and a *use* instruction may reference the same memory location. The guard is automatically produced by the substitution algorithm when replacing a memory reference with a replacement term in an expression (as was shown in Figure 3). Figure 6 shows the guarded expression produced due to potential aliasing from two *defs* to memory. An alias between a *use* followed by a *def* does not introduce a guard, since the *def* cannot affect the *use*.

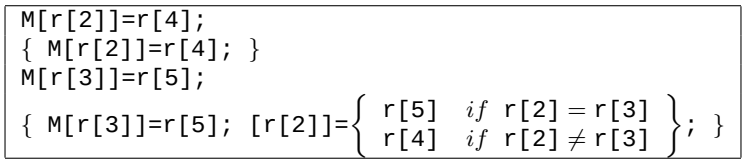


Figure 6: Potential Def-Def Alias

4.4 Proving Semantics Preserving Properties of Transformations with RTL Effects

Effects can be used to validate the semantics preserving property of transformations. An instance of a code-improving transformation is semantics preserving if the semantics of the transformed code are unchanged. Consider for example the following transformation:

```
r[0]=r[0]+1;
r[1]=r[0]-1;       r[0]:
M[r[1]]=M[r[2]];
r[0]=M[r[1]];
```

$\Rightarrow$

```
r[1]=r[0];         r[0]:
M[r[1]]=M[r[2]];
r[0]=M[r[1]];
```

The semantic effect of both code fragments is the same and identical to the effect

```
r[0]=M[r[2]]; r[1]=r[0]; M[r[0]]=M[r[2]];
```

Therefore, it is concluded that the transformation is semantics preserving.

This approach can be used to validate transformations on basic blocks comprising a consecutive sequence of RTL instructions. Transformations that may alter the flow of control however, require a more extensive analysis of the CFG and proof system. This is explained in the next section.

5 Validation of Transformations on Program Regions

This section presents the implementation details of the validation approach. The approach can handle transformations that change the control flow. To this end, a CFG analysis algorithm was implemented that determines the affected region of code. The algorithm also propagates postconditions and branch predicates down the CFG to determine the effects at the exit points in the region.

5.1 Overview

The automatic transformation-validating compilation system is illustrated in Figure 7. The system validates code-improving transformations in the *vpo* compiler [4]. The *vpo* compiler optimizes code using a CFG representation of the program [1], where the basic blocks of the CFG are populated with RTL instructions. The system analyzes the CFG for region changes after each optimizing transformation. The code that spans the affected region is identified and the semantic effects of the code are collected at each of the exit points of the region. The effects of the region before the transformation are compared to the effects of the region after the transformation. The transformation is valid if the normalized effects are found to be identical.

It is assumed that two regions are semantically identical if the functions in both regions are called in the same order and identical output data is produced in memory for each function call given any set of input data. The function call ordering requirement does not allow the validation of code-improving transformations that change the function call order. However, most compilers do not attempt to reorder function calls due to the

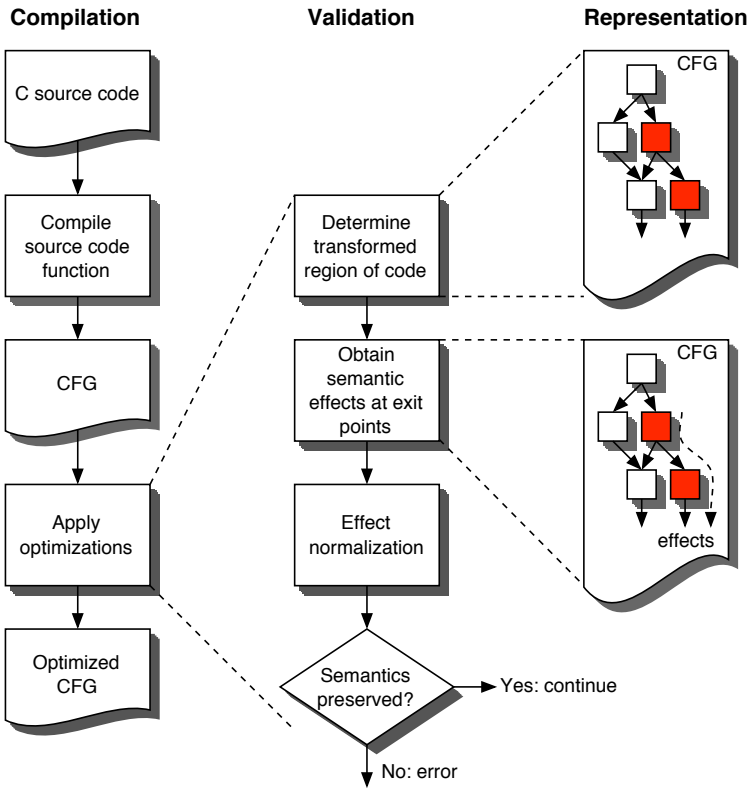


Figure 7: System Overview

presence of potential side effects in functions in procedural languages such as C. For example, *vpo* does not perform low-level code-optimizing transformations across function boundaries.

Note that the system proves the equivalence of the region of the program associated with the changes rather than showing the equivalence of the entire program representation. In [26] it was shown that the region that is changed by most code-improving compiler transformations is typically quite small, consisting of an average 5.9 RTL instructions for a set of benchmark programs.

The approach also supports the validation of hand-specified optimizations. This is currently accomplished with the the interactive VISTA system to edit RTL instructions and/or assembly instructions manually [26, 31].

5.2 Determining the Region of Code Associated with a Transformation

Determining the region of code that is associated with a code-improving transformation requires capturing the changes to the program representation caused by the transformation. Changes associated with a transformation are automatically detected by making a copy of the program representation before each code-improving transformation and comparing the program representation after the transformation with the copy. After identifying all of the basic blocks that have been changed, the closest block in the control-flow graph that dominates all of the modified blocks forms the root node of the region. This *dominating block* is the designated entry point for the region. The region consists of all instructions between the dominating block and the RTL instructions that have been modified. The region of code before the transformation will be re-

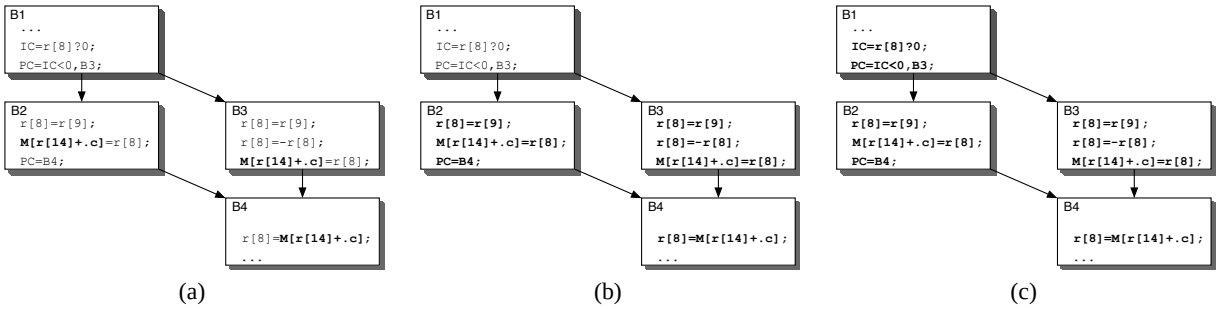


Figure 8: Example of Calculating the Extent of a Region

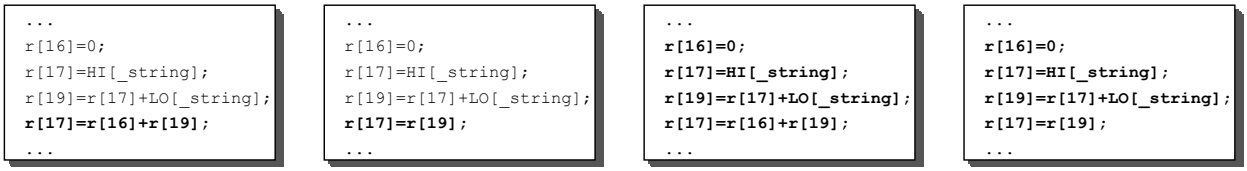


Figure 9: Example of Extending the Scope of a Region

ferred to as the *old region* and the region after the transformation will be referred to as the *new region*. These two regions are considered counterparts of each other since they should have the same semantic effects on the rest of the program. The effects of the old and new regions are considered semantically equivalent if they are identical at each exit point of the region. Note that the old and new regions need not have the same basic block structure. Only the dominating block and exit points of the two regions have to be identical.

The example shown in Figure 8 illustrates how the region detection algorithm works. Consider the program representation shown in Figure 8(a) depicting the state of the program before a register allocation transformation. The affected memory reference by the transformation is `M[r[14]+.c]` shown in boldface in the figure, where `.c` is a constant offset. This references a local variable that is replaced with a register in the transformation. The block that most closely dominates all blocks containing the modifications (blocks 2, 3, and 4) is block 1. The region consists of all RTL instructions between those that are changed and this dominating point, which are shown in boldface in Figure 8(b). Block 1 contains no RTL instructions that have been modified. As shown in Figure 8(c), its conditional branch is included in the region so conditions can be represented when transitions are made to blocks 2 and 3.

5.3 Extending the Region of Code Associated with a Transformation

There are cases when the extent of a region has to be recalculated. For instance, the points at which one region exits have to be identical to the exit points in its counterpart region. If an exit point in one region does not exist in its counterpart, then that exit point is added to its counterpart region and the extent of the region is recalculated.

A region may need to be extended when its effects are not identical as its counterpart region. This does not necessarily mean that the transformation is invalid. Rather, this may indicate that larger parts of the

code need to be included in the extent of the region to calculate the semantic effects. For instance, consider Figures 9(a) and 9(b). Only one change was detected, so the old and new regions initially consist of a single instruction (shown in boldface). Obviously, these two regions in isolation do not have the same effect. However, there is a reference to `r[16]` in one region that is not included in the other. If the effects of the two regions are not identical and there are more *uses* or *defs* of a specific register or a memory reference in one region, then the regions are extended to include an additional *def* of that register or memory reference. Figures 9(c) and 9(d) show the extension of the old and new regions to include the *def* of `r[16]`, which allows identical effects to be calculated for each region. The effects of Figure 9(c) and 9(d) are identical and equal to

```
r[16]=0; r[17]=HI[_string]+LO[_string]; r[19]=HI[_string]+LO[_string];
```

which provides proof that both code fragments are semantically identical. The `HI[]` and `LO[]` operators return the high part and low part of an address constant, respectively[2].

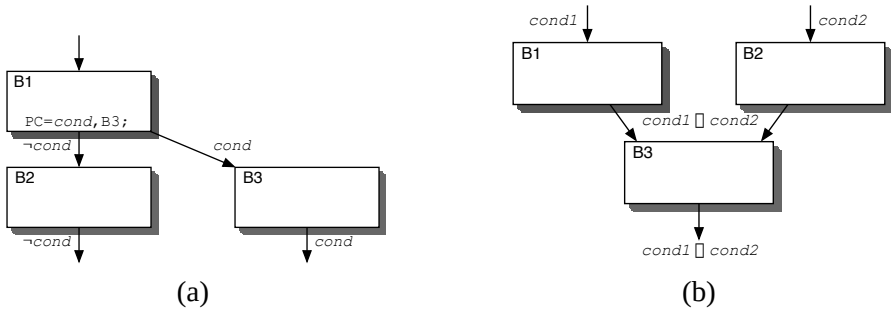
5.4 Calculating the Effects of a Region

Each region consists of a single entry point at the dominating block and one or more exit points. Semantic effects are calculated for each exit point from the region as follows:

- The liveness of registers and memory references in a region are calculated using a demand-driven data flow analysis method, as opposed to the traditional data flow analysis used in *vpo*. The demand-driven method caches and reuses previously computed *def* and *use* information obtained from earlier transformation and validation analyses stages to save time.
- Semantic effects are derived for each block in succession starting with the dominating block in the CFG and by working downwards until the exit points are reached. Back-edges in the CFG correspond to loops. A loop structure forms a connected component in the CFG and analyzed as such. This will be discussed in Section 5.6.
- To derive the semantic effects for each block in the CFG, the effects of the RTL instructions are merged, as was described previously.
- At a fork in the CFG the condition expression is propagated downward by predicating edges and nodes (basic blocks) with the branch condition, see Figure 10(a). The semantic effects derived from the basic blocks down the path are predicated with the conditions produced in these forks.
- A a join in the CFG the predicates of inbound edges are united to form a disjunction. This disjunction is propagated downward by predicating edges and nodes with the combined condition, see Figure 10(b).
- The process terminates when the effects are produced at all the exit points of the region.

The merging of effects of a region is accomplished using a directed acyclic graph (DAG) representation. A DAG effectively conserves storage, because common-subexpressions that occur multiple times in the effects of a region are stored only once. A DAG node is created for each right-hand side in an RTL assignment instruction and the expression is referenced by the right-hand side r-value of the assignment. In addition, the substitution algorithm (shown in Figure 3) was modified to process DAG nodes instead of symbolic

[2]The `HI[]` and `LO[]` operators are not included in the RTL grammar shown in Figure 1, because they are macros.



(a) (b)

Figure 10: Fork and Join Flow Predication

expressions. For example, the merging of `r[10]=r[8]-2;` with `r[11]=r[10]+r[10];` results in `r[10]`=n_1 and `r[11]`=n_2, where n_1 and n_2 are two DAG nodes with the bindings n_1=`r[8]-2` and the substitution n_2=n_1+n_1. Without a DAG representation for effects, merging can result in exponentially growing storage requirements.

The nodes in the final DAG (obtained after all effects have been merged) are marked when the node is used in an effect corresponding to an exit point of the region of code. The marking proceeds by recursively analyzing the expressions stored at the nodes in the DAG. Marking the DAG effectively eliminates unused expressions. Unused expressions typically are expressions assigned to registers or variables that are not live at the exit point of the region.

For each marked node visited in a postorder traversal of the DAG, the term stored at the marked node is simplified if not already simplified by rewriting the term into a canonical representation, where references in the term to other nodes are replaced by references to canonical representations of the terms in the nodes. When calculating the canonical representation of an effect, embedded references to canonical representations need not be simplified again. Finally, the effects at the exit points of the regions are simplified and the DAG node references are replaced by their canonical representations.

Effects are simplified using *Ctadel* to implement the rewrite rules shown in Figure 4 together with the usual simplification rules for arithmetic and logical expressions. *Ctadel* [25] is an extensible rule-based symbolic manipulation program implemented in SWI-Prolog [30]. The *vpo* compiler was modified by inserting calls to *Ctadel* to simplify effects. The expression simplification is applied in the address space of the *vpo* compiler by linking with the SWI-Prolog interpreter.

Consider the example program fragment shown in Figure 11(a). Figure 11(b) shows the CFG of the code with the edges predicated with the branch conditions. It is assumed that the compiler uses the following register assignments: `r[8]`≡`v`, `r[10]`≡`s`, and `r[11]`≡`inv`).

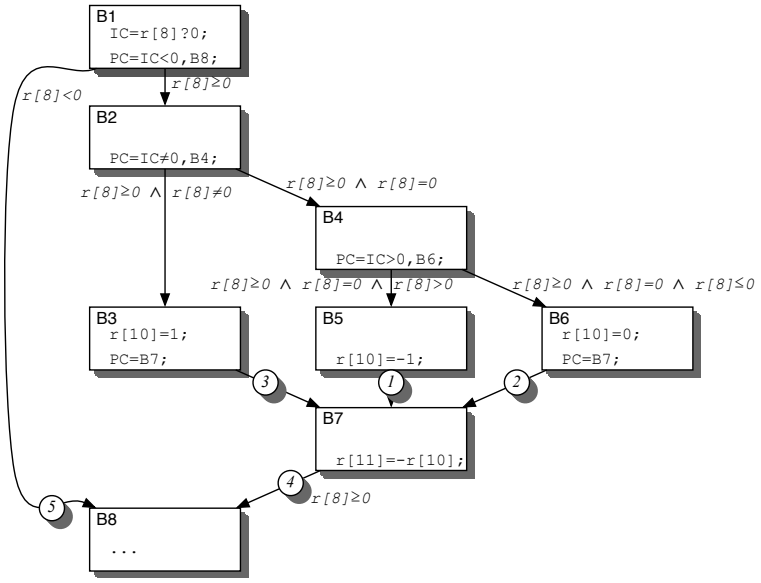
By following the paths through the branch structure, it is apparent that the statement `s=-1` is unreachable. Unreachable code is also referred to as *dead code* [1]. Dead code elimination is an optimizing compiler transformation that removes all blocks from a CFG that are unreachable. Therefore, the removal of block **B5** should not change the semantics of the program fragment. Indeed, the predicate at the edge from block **B4** to **B5** indicates that block **B5** is not reachable. Therefore, the effects collected in **B5** are predicated with *false* and not further propagated.

The transitions (1)–(5) in the CFG are of particular interest the derivation of the edge predicates will be discussed in more detail:

1. At transition (1) the semantic effect from block **B5** is $\mathtt{r}[10] = \{-1\ if\ \mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] = 0 \wedge \mathtt{r}[8] > 0\}$

```
if (v>=0)
{   if (v!=0)
        s=1;
    else if (v<=0)
        s=0;
    else
        s=-1;
    inv=-s;
}
```

(a)



(b)

Figure 11: Example

2. At transition (2) the semantic effect from block **B6** is $\mathtt{r}[10] = \{0\ if\ \mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] = 0 \wedge \mathtt{r}[8] \leq 0\}$

3. At transition (3) the semantic effect from block **B3** is $\mathtt{r}[10] = \{1\ if\ \mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] \neq 0\}$

4. Combining the effects of the blocks **B5**, **B6**, and **B3** at the join point gives

$$\mathtt{r}[10] = \left\{ \begin{array}{ll} 1 & if\ \mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] \neq 0 \\ 0 & if\ \mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] = 0 \wedge \mathtt{r}[8] \leq 0 \\ -1 & if\ \mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] = 0 \wedge \mathtt{r}[8] > 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} 1 & if\ \mathtt{r}[8] > 0 \\ 0 & if\ \mathtt{r}[8] = 0 \end{array} \right\}$$

5. Merging these effects with the effects of block **B7** yields

$$\begin{array}{rcl} \mathtt{r}[10] & = & \left\{ \begin{array}{ll} 1 & if\ \mathtt{r}[8] > 0 \\ 0 & if\ \mathtt{r}[8] = 0 \end{array} \right\} \\ \mathtt{r}[11] & = & -\left\{ \begin{array}{ll} 1 & if\ \mathtt{r}[8] > 0 \\ 0 & if\ \mathtt{r}[8] = 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} -1 & if\ \mathtt{r}[8] > 0 \\ 0 & if\ \mathtt{r}[8] = 0 \end{array} \right\} \end{array}$$

6. Merging the (empty) effects at transition (5) with the effects at transition (4) we obtain the effects of the region of code

$$\begin{array}{rcl} \mathtt{r}[10] & = & \left\{ \begin{array}{ll} \left\{ \begin{array}{ll} 1 & if\ \mathtt{r}[8] > 0 \\ 0 & if\ \mathtt{r}[8] = 0 \end{array} \right\} & if\ \mathtt{r}[8] \geq 0 \\ \mathtt{r}[10] & if\ \mathtt{r}[8] < 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} 1 & if\ \mathtt{r}[8] > 0 \\ 0 & if\ \mathtt{r}[8] = 0 \\ \mathtt{r}[10] & if\ r[8] < 0 \end{array} \right\} \\ \mathtt{r}[11] & = & \left\{ \begin{array}{ll} \left\{ \begin{array}{ll} -1 & if\ \mathtt{r}[8] > 0 \\ 0 & if\ \mathtt{r}[8] = 0 \end{array} \right\} & if\ \mathtt{r}[8] \geq 0 \\ \mathtt{r}[11] & if\ \mathtt{r}[8] < 0 \end{array} \right\} \stackrel{\text{simplify}}{=} \left\{ \begin{array}{ll} -1 & if\ \mathtt{r}[8] > 0 \\ 0 & if\ \mathtt{r}[8] = 0 \\ \mathtt{r}[11] & if\ r[8] < 0 \end{array} \right\} \end{array}$$

where the guard condition $\mathtt{r}[8] \geq 0$ is derived by forming the disjunction of the guard conditions on the incoming edges to block **B7**, which is the simplified form of

$$(\mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] \neq 0) \vee (\mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] = 0 \wedge \mathtt{r}[8] \leq 0) \vee (\mathtt{r}[8] \geq 0 \wedge \mathtt{r}[8] = 0 \wedge \mathtt{r}[8] > 0)$$

The final effects to `r[10]` and `r[11]` in step 6 accurately describe the semantics of the region of code. The semantic effect of the region of code before and after a dead-code elimination transformation is applied is unchanged (not shown).

5.5 Representing Effects from Function Calls

The combined effects on registers and memory by a function call can be assumed to be the same when the function arguments are unaltered and the function calls take place in the same order. To accurately represent the effects of function calls on the state of registers and (global) memory references, it was decided to use an RTL assignment-based notation to describe effects on registers and memory references. The effects representation conservatively assumes that all registers and memory references are potentially changed. For each live register and memory reference a function call is represented by a sequence of RTL instructions of the form

L=func(L, *label*);

where L is an l-value and *label* refers to the basic block (of the original old region) that contains the function call. In this way, functions are viewed as black-box transformers of l-values. The block labeling ensures that transformations that change the function call order will be flagged as invalid.

5.6 Representing Effects from Loops

A region may span multiple basic blocks that cross loop nesting levels. Merging the effects across loop nesting levels requires calculating the effects of an entire loop. One issue that must be addressed is how to represent a recurrence, which involves the use of a variable or register that is set on a previous loop iteration. An induction variable is one example of a simple recurrence. We represent a recurrence as a recursive function using the following notation. The *label* distinguishes the loop in which the recurrence occurs. The *newValue* represents the next value of the recurrence. References to *w* in the *newValue* represent the previous value of the recurrence. The *initialValue* is the initial value of the recurrence. The *condition* indicates when the recurrence is no longer applied. Thus, this notation is used to represent a sequential ordering of effects, where each instance represents the effect on a different iteration of a loop.

$$\mathtt{y}(label, initialValue, newValue)\ \mathtt{until}\ condition$$

We define the semantics of the recurrence $\mathtt{y}(label, initialValue, newValue)$ by defining function F as

$$F = \lambda f.\lambda i.\text{if } i = 0 \text{ then } initialValue \text{ else } (\lambda w.newValue)\,(f\,(i-1))$$

The semantics of y is defined as the application of the fixpoint **Y** combinator to F, which results in a function that given an iteration number i ($i \geq 0$) returns the value of the recurrences at that iteration. For example, the value of the recurrence $\mathtt{y}(\mathtt{B2}, 1, w+1)$ at iteration 10 is

$$\mathbf{Y}(\lambda f.\lambda i.\text{if } i = 0 \text{ then } 1 \text{ else } (\lambda w.w+1\rangle)\,(f\,(i-1))\,10 = 11$$

1. Effects of **B1**:
   `r[10]=0; r[11]=HI[_a]; r[12]=HI[_n];`
   Effects of **B2** after merging with effects of **B1**:
   `M[r[10]<<2+r[11]+LO[_a]]=0; r[10]=r[10]+1;`

2. Effects of **B1**:
   `r[10]=0; r[11]=HI[_a]; r[12]=HI[_n];`
   Introducing a recursive function for recurrences in effects of **B2**:
   ```
   M[(y(B2,r[10],w+1) until
      y(B2,r[10]+1,w+1)≥M[r[12]+LO[_n]])<<2+r[11]+LO[_a]]=0;
   r[10]=y(B2,r[10],w+1) until
         y(B2,r[10]+1,w+1)≥M[r[12]+LO[_n]];
   ```

3. After merging the effects of **B1** into effects of **B2**:
   ```
   M[(y(B2,0,w+1) until
     y(B2,0+1,w+1)≥M[HI[_n]+LO[_n]])<<2+HI[_a]+LO[_a]]=0;
   r[10]=y(B2,0,w+1) until y(B2,0+1,w+1)≥M[HI[_n]+LO[_n]];
   ```

4. After simplification:
   ```
   M[(y(B2,0,w+1) until y(B2,1,w+1)≥M[_n])*4+_a]=0;
   r[10]=y(B2,0,w+1) until y(B2,1,w+1)≥M[_n];
   ```

5. After changing the recursive functions to sequences:
   ```
   M[({B2,0,1} until y(B2,1,w+1)≥M[_n])*4+_a]=0;
   r[10]={B2,0,1} until y(B2,1,w+1)≥M[_n];
   ```

6. After adjusting initial value and increment of the first sequence:
   ```
   M[({B2,_a,4} until y(B2,1,w+1)≥M[_n])]=0;
   r[10]={B2,0,1} until y(B2,1,w+1)≥M[_n];
   ```

7. After detecting that `r[10]` is not used after the loop:
   ```
   M[({B2,_a,4} until y(B2,1,w+1)≥M[_n])]=0;
   ```

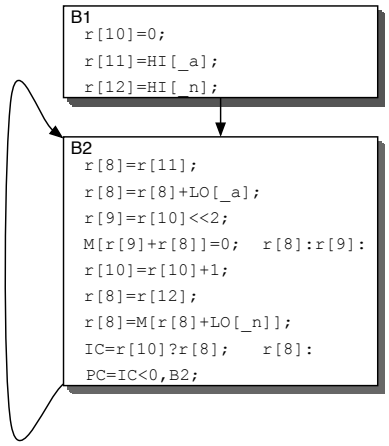


Figure 12: Example of Calculating Effects for a Loop

A *sequence* can be used to represent recursive functions when the *newValue* is obtained by incrementing the current value, which is the case for basic induction variables. We adopt a notation that is similar to the notation used for chains of recurrences [2, 27, 28] (CRs). Each sequence has the following form, which is similar to a recursive function. Unlike recursive functions, a number of algebraic operations can be applied to these sequences.

$$\{label, initialValue, increment\}\ \texttt{until}\ condition$$

Figure 12 depicts an example of representing the effects of a loop. In this example, the loop is about to be modified by a loop strength reduction transformation. Step 1 shows the effects of the preheader and loop blocks after merging the effects of the blocks separately. Step 2 shows the effects of the loop block after replacing all uses of `r[10]` with a recursive function since the assignment `r[10]=r[10]+1;` forms a recurrence. Note that the *condition* of the recursive function is the exit condition of the loop. Step 3 shows the effects after merging the preheader block with the loop block. The register `r[10]`, unlike registers `r[11]` and `r[12]`, is not loop invariant. However, the only explicit uses of `r[10]` in the loop after applying the recursive functions is in the *initialValue* of these functions. Thus, assignments before a loop

Program	Description	Num Trans	Validated	Region Size	Overhead
ackerman	benchmark that performs recursive function calls	89	100.0%	3.18	13.64
arraymerge	benchmark that merges two sorted arrays	483	89.2%	4.23	63.89
banner	poster generator	385	90.6%	5.42	34.13
bubblesort	benchmark that performs a bubblesort on an array	342	85.4%	6.10	34.37
cal	calendar generator	790	91.1%	5.16	105.64
head	displays the first few lines of files	302	89.4%	8.42	152.64
matmult	multiplies 2 square matrices	312	89.7%	5.55	28.97
puzzle	benchmark that solves a puzzle	1928	78.5%	5.85	128.98
queens	eight queens problem	296	85.8%	6.79	73.65
sieve	finds all prime numbers between 3 and 16383	217	80.6%	6.85	21.90
sum	prints the checksum and block count for a file	235	91.9%	8.62	195.19
uniq	report or filter out repeated lines in a file	519	91.1%	4.21	163.26
average		492	88.6%	5.87	84.64

Table 1: Benchmarks

can be successfully merged into a loop after representing recurrences with recursive functions. Likewise, effects after a loop will be guarded by the exit condition, which will allow merging of effects in a loop with effects after the loop. Step 4 shows the effects after simplification. Constant expressions are simplified, the sum of the high and low portions of a value is replaced with the value itself, and left shifts by a constant are replaced by multiplies. Step 5 changes the recursive functions to sequences. Step 6 adjusts the first sequence by a factor and an offset. Finally, step 7 deletes the assignment to `r[10]` since `r[10]` is not used after the loop.

Recurrences are a concise representation of a loop iteration. The underlying semantics of recurrences can be best viewed as a representation that enables us to unroll the iterations of a loop until the condition is satisfied thereby obtaining the semantic effects of each loop iteration. Multiple recurrences refering to the same loop are unrolled in parallel, hence the use of a *label* to refer to a loop. For instance, the effect of step 7 in Figure 12 is `M[({B2,_a,4} until y(B2,1,w+1)≥M[_n])]=0;` which represents the combined execution of all iterations

$$
\begin{aligned}
\texttt{M[\_a]} &= (0 \ \textit{if} \ 1 < \texttt{M[\_n]});\\
\texttt{M[\_a+4]} &= (0 \ \textit{if} \ 2 < \texttt{M[\_n]});\\
&\vdots\\
\texttt{M[\_a+36]} &= (0 \ \textit{if} \ 10 < \texttt{M[\_n]});
\end{aligned}
$$

where the value of `M[_n]` is assumed to be 10 and is not aliased with an element of the array `_a`.

6 Results

Table 1 shows the results on a number of test programs that were compiled while validating code-improving transformations in *vpo*. The third column indicates the number of improving transformations that were applied during the compilation of the program. The fourth column represents the percentage of the transformations that were attempted to be validated. The compiler did not attempt to validate the transformations if the region of the transformation extends across one or more loop boundaries. For those attempted, the success rate was 100% validation as can be expected from a high-quality compiler. Therefore, no false alarms were raised for the transformations whose regions do not extend across one or more loop boundary. The reason for not attempting to validate those is that the loop analysis (as shown in Section 5.6) has not yet been fully implemented, due to a tighter interaction required between the *vpo* compiler and the *Ctadel* system to

handle the loop-based effects simplification. The fifth column represents the average static number of instructions for each region associated with all code-improving transformations during the compilation. The final column denotes the ratio of compilation times when validating programs versus a normal compilation.

The types of transformations in the *vpo* compiler that were validated using the system include *algebraic simplification of expressions*, *basic block reordering*, *branch chaining*, *common subexpression elimination*, *constant folding*, *constant propagation*, *unreachable code elimination*, *dead store elimination*, *evaluation order determination*, *filling delay slots*, *induction variable removal*, *instruction selection*, *jump minimization*, *register allocation*, *strength reduction*, and *useless jump elimination*. These and similar transformations are typically applied by a low-level code-improving compiler.

The use of an interpretive Prolog system to simplify effects did impact the speed of the validation process. However, an overhead of about two orders of magnitude would probably be acceptable, as compared to the cost of not detecting potential errors. Note that a user can select a subset of transformations (e.g. ones recently implemented) to be validated. In addition, validation would not be performed on every compilation.

7 Conclusions

This paper described a general approach for the automatic validation of code-improving transformations on low-level code. First, the region in the program representation associated with the changes caused by a code-improving transformation is identified. Second, the effects of the region before and after the transformation are calculated. Third, a set of simplification rules are applied to normalize the form of these effects. Finally, the effects of the region before and after the transformation are compared. If the two sets of effects are identical, then the transformation is deemed valid. It was shown that it is feasible to use this approach to validate many conventional code-improving transformations.

Validating code-improving transformations has many potential benefits. Validation provides greater assurance of correct compilation of programs, which is important since software is being used as a component of an increasing number of critical systems. The time spent by compiler writers to detect errors can be dramatically reduced since the transformations that do not preserve the semantics of the program representation are identified during the compilation. Finally, validation of hand-specified transformations on assembly code can be performed, which can assist programmers of embedded systems.

There has also been progress in proving type, memory safeness, and other related properties of a compilation rather than the equivalence of source and target programs [22, 18, 19]. Proving these types of properties is important and these techniques could be used in conjunction with the approach described in this paper.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[2] O. Bachmann, P.S. Wang, and E.V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computing*, pages 242–249, Oxford, 1994. ACM.

[3] Lionel Bening and Harry Foster. *Principes of Verifiable RTL Design*. Kluwer Academic, 2001.

[4] M. E. Benitez and J. W. Davidson. A Portable Global Optimizer and Linker. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 329–338, June 1988.

[5] J.A. Bergstra, T.B. Dinish, J. Field, and J. Heering. A complete transformational toolkit for compilers. In H.R. Nielson, editor, 6^{th} *European Symposium on Programming (ESOP'96), LNCS 1058*, Linköping, Sweden, April 1996. Springer. Also Technical Report CS-R9601, National Research Institute for Mathematics and Computer Science (CWI), The Netherlands.

[6] M. Boyd and D. Whalley. Isolation and Analysis of Optimization Errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–35, June 1993.

[7] M. Boyd and D. Whalley. Graphical Visualization of Compiler Optimizations. *Journal of Programming Languages*, 3:69–94, 1995.

[8] A. Cimatti and et. al. A Provably Correct Embedded Verifier for the Certification of Safety Critical Software. In *International Conference on Computer Aided Verification*, pages 202–213, June 1997.

[9] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[10] C. Dow, S. Chang, and M. Soffa. A Visualization System for Parallelizing Programs. In *Proceedings of Supercomputing*, pages 194–203, 1992.

[11] P. Dybjer. Using Domain Algebras to Prove the Correctness of a Compiler. *Lecture Notes in Computer Science*, 182:329–338, 1986.

[12] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a Verified Implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.

[13] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, October 1969.

[14] S. Horwitz. Identifying the Semantic and Textual Differences between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[15] C. Jaramillo, R. Gupta, and M. L. Soffa. Capturing the Effects of Code Improving Transformations. In *International Conference on Parallel Architecture and Compilation Techniques*, pages 118–123, October 1998.

[16] J. Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[17] F. Morris. Advice on Structuring Compilers and Proving Them Correct. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 144–152, 1973.

[18] G. Necula. Proof-Carrying Code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[19] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[20] G. C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the SIGPLAN '00 Symposium on Programming Language Design and Implementation*, pages 83–94, June 2000.

[21] M. Rinard and D. Marinov. Credible Compilation with Pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verfication*, 1999.

[22] D. Tarditi, J. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[23] J. Thatcher, E. Wagner, and J. Wright. More on Advice on Structuring Compilers and Proving Them Correct. In *Proceedings of a Workshop on Semantics-Directed Compiler Generation*, pages 165–188, 1994.

[24] R. van Engelen, D. Whalley, and X. Yuan. Automatic validation of code-improving transformations. In *ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, 2000.

[25] R.A. van Engelen, L. Wolters, and G. Cats. CTADEL: A generator of multi-platform high performance codes for PDE-based scientific applications. In 10^{th} *ACM International Conference on Supercomputing*, pages 86–93, New York, 1996. ACM Press.

[26] Robert van Engelen, David Whalley, and Xin Yuan. Validation of code-improving transformations for embedded systems. In *proceedings of the 8th ACM Symposium on Applied Computing SAC 2003*, pages 684–691, Melbourne Florida, March 2003.

[27] Robert A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proc. of the ETAPS Conference on Compiler Construction 2001, LNCS 2027*, pages 118–132, 2001.

[28] Robert A. van Engelen and Kyle Gallivan. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems (IWIA) 2001*, pages 80–89, Maui, Hawaii, 2001.

[29] D. Whalley. Automatic Isolation of Compiler Errors. *ACM Transactions on Programming Languages and Systems*, 16:1648–1659, 1994.

[30] J. Wielemaker. *SWI-Prolog Reference Manual*. University of Amsterdam, 1995. Available by anonymous ftp from `swi.psy.uva.nl`.

[31] Wankang Zhao, Baosheng Cai, David Whalley, Mark Bailey, Robert van Engelen, Xin Yuan, Jason Hiser, Jack Davidson, Kyle Gallivan, and Douglas Jones. VISTA: A system for interactive code improvement. In *proceedings of the LCTES conference*, 2002.