Verification of Compiler Optimization Using Temporal Logic by Checking Value Equality Difference

Ling Fang [1,2] Masataka Sassa [3]

Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology,
2-12-1, O-okayama, Meguro-ku, Tokyo 152-8552, Japan

Abstract

Optimization is a very important phase of compilation. It can improve the performance of a program by a large factor. However, many recent optimizers are complex, which may compromise program correctness. It is essential that the compiler optimizer is implemented without changing the semantics of a program. Guaranteeing the correctness of optimization for realistic programs is still an open problem. In this paper, we propose a technique for validating the optimization transformation of the program by checking if the optimization transformations are equivalent transformations. First, we define seven Computational Tree Logic (CTL) formulas for judging the correctness of seven types of transformation. Then, we compare the intermediate programs before and after optimization and extract the differences. After analysis and modeling, every transformation is checked against the corresponding CTL formula via model checking. We do not need to know the details of the optimization algorithms. We applied this technique to the optimizer in the COmpiler INfraStructure (COINS) compiler. It worked with great efficiency, and found several bugs and ambiguous transformations.

Key words: compiler optimization, equivalent transformations, temporal logic, model checking

1 Introduction

1.1 Background

Program optimization in compilers is an important technology and is being investigated actively. Optimizations improve the execution speed or the size of a program. A compiler optimizer must not change the semantics of the program. Generally, however, the compiler optimizer is very complex, and

bugs may be introduced at various stages of optimization, including algorithm design, implementation design, and coding. The bug may let the optimized program appear to run normally, but its behavior may be incorrect and the user cannot judge if it is correct or not. Also, it may not cause semantic errors, but may add redundancy to the program, contradicting the purpose of optimization. Even if an unexpected value or behavior occurs, the cause of the bug is not ascertained easily. Therefore, technology that can find the causes of optimizer bugs is very important. Previous work that has improved the reliability of compiler optimizers includes these categories:

 (i) Validating the optimizers themselves,

 (ii) Validating a test program optimized by the optimizer by checking if the transformation is correct,

(iii) Validating a test program's execution by checking if corresponding variable values are the same before and after optimization.

Category (i) includes work by Lacey et al. [12], and Lerner et al. [13]. This research guarantees theoretical strictness by defining a domain language that can be proven. However, these methods can deal only with simple optimizations that are unlikely to introduce bugs and they can only validate an optimizer written in the special domain language.

Category (ii) includes work by Necula et al. [14], and Sassa et al. [17]. Necula et al.'s work assigns annotation pairs to source and target program points that must be equivalent before and after optimization, and performs an abstract symbolic execution to validate if the equivalence is satisfied. It is very practical and can be used in any optimization. However, the validation is a kind of assertion and it sometimes fails. Moreover, many false alarms that are not bugs are reported. Sassa et al.'s research marks the test program's points by assertions that are related to the optimization algorithm while executing the optimization, and checks the relations of these assertions via model checking. Because this technique relies on the optimization algorithm, the user (compiler writer or tester) must understand the algorithm sufficiently to find where to mark the assertions and must enhance the optimizer with aspect-oriented GluonJ [3]. Many check formulas must be described for each phase of the optimizer. Sometimes, points cannot be marked because the relations are too complex to describe. Neither [14] nor [17] preserve the entire semantic equivalence, and accuracy decreases for complex optimizations.

Category (iii) includes Jaramillo et al.'s work [10]. Their method checks if corresponding variable values in the program are equivalent before and after optimization. The work is theoretically weak because it can only validate a concrete execution of a concrete program. It is difficult to assert that the optimization is correct if the assertion is based only on the values of some variables in a test program being executed. The method can work only after an executable program is generated. Moreover, the execution trace is very large.

1.2   Outline

Our method belongs to research category (ii) described in Section 1.1. It automatically extracts differences in the intermediate programs before and after optimization, and checks whether the transformations are semantically equivalent. No execution is necessary. The advantages of our technique are as follows.

- It is strict because all transformations are validated. Very complex optimizations, such as CSTP [4] and PREQP [5], can be accurately verified.

- It does not rely on the algorithm or the implementation of the optimization. Therefore, It can be applied to any optimizers without needing information about the optimizer, and the user does not need to understand the algorithm of the optimizer or its implementation, or enhance the optimizer. No execution is necessary. Formalization is very easy.

- The validation time is practical and it is easy to ascertain the cause when a bug is found. Moreover, it is the first time that redundancy, in addition to semantic errors, can be reported.

We have applied our method to the optimizer implemented in the widely used COINS compiler [5] and have found several unknown bugs. Two of them are semantic errors that have embarrassed the COINS development group for a long time.

2   SSA Form and Optimization in SSA Form

Here, we introduce briefly the SSA form [8] and optimization in SSA form. We use SSA form because the comparison of programs before and after optimization is very easy if they are in SSA form.

2.1   SSA form

The SSA form is a form of program in which the definition of a variable occurs only once in the program. The definition that reaches to the use of the variable is unique, so it is advantageous for the optimization of the program. Figure 1 are two examples of the normal forms and Figure 2 are their SSA forms. The ϕ function is a virtual function inserted at the place where the definitions of the same variables reach a control flow merge point. In Figure 2 (right), $x3 = \phi(x2 : L2, x1 : L1)$ is an example of a ϕ function. It means that $x2$ is assigned to $x3$ when coming from $L2$, and $x1$ is assigned to $x3$ when coming from $L1$. Moreover, after the substitution $x3 = \phi(x2 : L2, x1 : L1)$, all uses of x have been rewritten to $x3$.

---

[4]   Constant Folding and Propagation with Conditional Branches [22]
[5]   Partial Redundancy Elimination with Question Propagation [20]
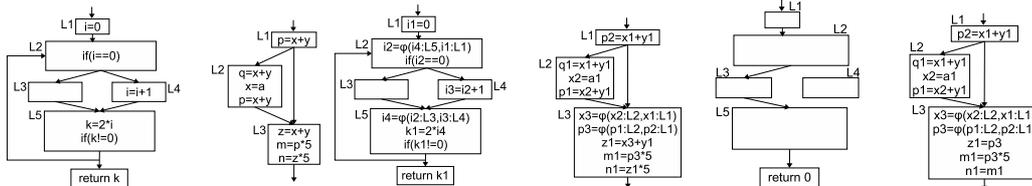
Fig. 1. Normal form

Fig. 2. SSA form

Fig. 3. Optimization in SSA form

2.2 Optimization in SSA form

Optimizations improve the execution speed or the size of a program. At the very least, the optimizer must preserve the semantics of the original program. Moreover, it should not perform useless transformations or, worse, add redundancy to the optimized program.

Although using data flow equations in the control flow graph is the most general approach, there is also a kind of optimization, named CSTP [22], that cannot be dealt with by this method because it requires abstract interpretation [7] of the program. CSTP is a powerful optimization in the SSA form. Figure 2 (left) and Figure 3 (left) are the program before and after CSTP. $i1$ is always 0 and block $L4$ is unreachable, and these facts are clarified by sequentially tracing the blocks from their entrance by abstract interpretation. CSTP cannot be solved by data flow equations, which have the calculating power of a μ calculation [21]. The methods of Lacey et al. [12] and Lerner et al. [13] cannot deal with this optimization. Sassa et al. [17] indicate that CSTP is verified, but we believe that the verification is circular reasoning, and the execution of their verification system falls into an endless loop when actually validating CSTP.

Although the SSA form has many advantages, because in Figure 2 (right), $L3 : z1 = x3 + y1$, $L1 : p2 = x1 + y1$ and $L2 : p1 = x2 + y1$ are different, the fact that $z = x + y$ is a redundant calculation in Figure 1 (right) is lost after the SSA transformation. PREQP is an optimization that can identify this type of redundancy and then eliminate it. However, it is sufficiently complex that no previous work has been able to verify it. Figure 2 (right) and Figure 3 (right) show the program before and after PREQP, where the right-hand sides of $n1$ and $z1$ are rewritten after PREQP.

3 Temporal Logic

Computational Tree Logic (CTL) [21] can describe the properties of endless branch paths in Kripke structures. We adopt it in our research because it is very suitable for describing the properties of a program if the control flow graph (CFG)[1] corresponds to a Kripke structure and the abstract interpretation corresponds to endless branch paths. In this section, we present the syntax and semantics of CTL. Figure 4 is an example of a Kripke structure (a) and its CTL (infinite) trees (b) for S_0.

In this paper, the order of time is the direction of the directed link in a Kripke structure.

3.1 Syntax and semantics of CTL

Syntax

$$\phi ::= true \mid false \mid \alpha$$
$$\mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$$
$$\mid E\psi \mid A\psi$$
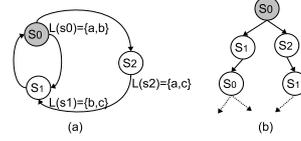$$\psi ::= X\phi \mid \phi \, U \, \phi$$

Fig. 4. Kripke structure (a) and its CTL (infinite) trees (b) for S_0

The standard abbreviations, e.g., $F\phi \equiv true \, U \, \phi$, $G\phi \equiv \neg F\neg\phi$, $\phi 1 \, R \, \phi 2 \equiv \neg(\neg\phi 1 \, U\neg \, \phi 2)$ can also be used.

Semantics

The semantics of CTL are given via the Kripke structure. A Kripke structure K is a triple (N, R, L). N is a set of states, $R \subseteq N \times N$ is a transition relation, and $L : N \rightarrow 2^\alpha$ is a function that maps each state to a set of primitive propositions that are true in that state.

A path from n_0 in K is the infinite sequence of states $\pi = (n_0, n_1, \ldots)$ such that $\forall i \geq 0 : (n_i, n_{i+1}) \in R$.

$K, n \models \phi$ denotes that the value of logical formula ϕ is true in state n in a Kripke structure K. The "K" can be omitted if it is obvious. The relation $\models$ is defined as follows.

State Formula:

    $n \models true$ iff $true$
    $n \models false$ iff $false$
    $n \models \alpha$ iff $\alpha \in L(n)$
    $n \models \neg\phi$ iff not $n \models \phi$
    $n \models \phi_1 \wedge \phi_2$ iff $n \models \phi_1$ and $n \models \phi_2$
    $n \models \phi_1 \vee \phi_2$ iff $n \models \phi_1$ or $n \models \phi_2$
    $n \models E\psi$ iff $\exists path(n = n_0 \rightarrow n_1 \rightarrow n_2 \ldots) : (n_i)_{i \geq 0} \models \psi$
    $n \models A\psi$ iff $\forall path(n = n_0 \rightarrow n_1 \rightarrow n_2 \ldots) : (n_i)_{i \geq 0} \models \psi$

Path Formula:

    $(n_i)_{i \geq 0} \models X\phi$ iff $n_1 \models \phi$
    $(n_i)_{i \geq 0} \models \phi_1 U \phi_2$ iff $\exists j \geq 0[n_j \models \phi_2 \wedge \forall i : [0 \leq i < j \implies n_i \models \phi_1]]$

We use symbols $\overleftarrow{A}$ and $\overleftarrow{E}$ to mean the opposite direction for a path. They are completely symmetrical with respect to the positive direction symbols A and E.

Circular reasoning

Circular reasoning must be avoided when validating by logic. To avoid circular reasoning, the condition must appear earlier than the conclusion.

4 Proposed Method

The details of our method will be described in this section.

4.1 Outline of the proposed method

In this paper, we propose a method that checks if the transformations in the optimization are semantically equivalent transformations, i.e., the optimization preserves the semantics of the original program. To achieve this, we use model checking, as follows:

- First, we define seven CTL formulas for judging the correctness of seven types of transformation.
- Next, we extract the differences in the intermediate programs before and after optimization, and create a model based on the intermediate program before its optimization, that reflects the transformations.
- Then we check if every transformation satisfies the corresponding CTL formula in the model, and report a bug if a CTL formula cannot be satisfied.
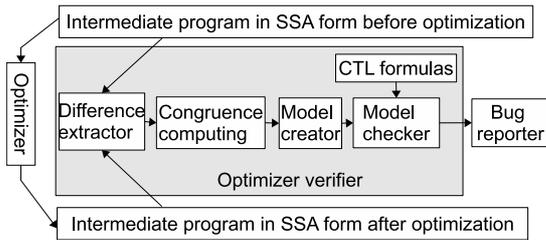
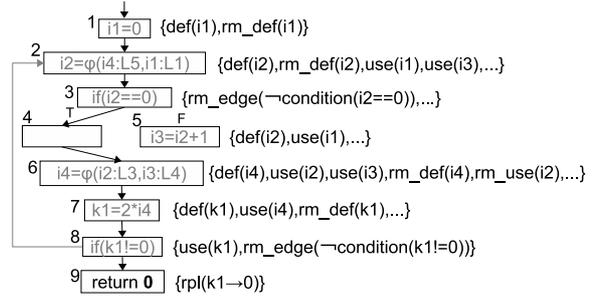Figure 5 shows the outline of our proposal.

Fig. 5. Outline of the system

Fig. 6. Model

For ease of understanding, we first explain how to create the model, then describe the CTL formulas for model checking while explaining the extraction of transformations.

4.2 Creation of the model

As described in Section 3, the semantics of CTL is given by the Kripke structure, which is a set $\{N, R, L\}$. It is necessary to construct a model corresponding to a Kripke structure if temporal logic is to apply. The models before and after optimization are assumed to be $M = \{N, R, L\}$, $M' = \{N', R', L'\}$, and $\tilde{M} = \{\tilde{N}, \tilde{R}, \tilde{L}\}$ is the model based on M that reflects the transformation of M'. $\tilde{M}$ will be the target model used for model checking. Figure 6 is a simple example of the model $\tilde{M}$ of the program in Figure 2 (left). In the following paragraphs, we will sometimes use a basic block for ease of explanation, although the unit forming the nodes of the model is the statement.

6

There are seven types of transformation of a program. The details will be described in Section 4.3, but we can classify them approximately as statement deletion, statement replacement, statement insertion, and branch deletion. After the transformations are extracted, the model can be created by the following steps.

- Create M and M' based on the program's CFG before and after optimization. The properties of definition-use relation etc. are added to each node as primitive propositions. For example, for the node of the statement $i3 = i2 + 1$, primitive propositions $def(i3)$ and $use(i2)$ are added to the node.

- Compare M and M' and create the model $\tilde{M}$, which is based on M and reflects the transformations of M'. For example, $rm\_def(i3)$ and $rm\_use(i2)$ are added to the node because, in the model M', statement $i3 = i2 + 1$ is removed after optimization.

Primitive propositions $\tilde{L}$

The primitive propositions in Table 1 describe properties such as definition or usage of variable and their transformations.

$\tilde{L}(n) =$
$\cup$	{ $use(\texttt{X})$	variable $\texttt{X}$ is used at node n }
$\cup$	{ $def(\texttt{X})$	variable $\texttt{X}$ is defined at node n }
$\cup$	{ $trans(\texttt{X})$	variable $\texttt{X}$ is not changed at node n }
$\cup$	{ $del\_def(\texttt{X})$	variable $\texttt{X}$ is defined at node n but deleted after optimization}
$\cup$	{ $ins\_def(\texttt{X})$	variable $\texttt{X}$ is inserted at node n after optimization}
$\cup$	{ $rm\_branch(\texttt{E})$	conditional statement branching to edge $\texttt{E}$ is deleted}
$\cup$	{ $equal(\texttt{X},\texttt{Y})$	$\texttt{X}$ and $\texttt{Y}$ belong to the same congruence set}
$\cup$	{ $\dots$	$\dots$}

Table 1
Definition of $\tilde{L}(n)$

States set $\tilde{S}$

For replaced or deleted statements, the nodes remain with primitive propositions such as $rpl(v1 \rightarrow v2)$ or $rm\_def(v1)$ being added to the nodes. For an inserted assignment, the node is added to the model with a primitive proposition such as $ins\_def(v1)$ indicating that the node has been inserted. Therefore, $\tilde{S} = S \cup \Sigma node'$, with $node'$ being a statement inserted after optimization. These are often temporal variables.

Transition relation $\tilde{R}$

The following is the transition relation $\tilde{R}$. We use Figure 7 to explain it. Here, (1) means an unchanged relation after optimization, such as $n_2 \rightarrow n_3$. (2) means the preservation of a relation because a removed node remains, such as $n_3 \rightarrow n_5$. (3) means a changed relation because of inserted nodes, such as

$n_1 \rightarrow n_2$ iff
$\vee$ $(n_1 \rightarrow n_2 \in R \wedge n_1 \rightarrow n_2 \in R') \ldots \ldots (1)$
$\vee$ $(n_1 \rightarrow n_2 \in R \wedge n_1 \in rm\_node \vee n_2 \in rm\_node) \ldots \ldots (2)$
$\vee$ $(n_1 \rightarrow n_2 \in R \wedge n_1 \rightarrow n_2 \notin R' \wedge n_1 \rightarrow n'_1 \rightarrow \ldots \rightarrow n'_n \rightarrow n_2 \in R') \ldots \ldots (3)$
$\vee$ $(n_1 \rightarrow n_2 \notin rm\_branch) \ldots \ldots (4)$

$n_1 \rightarrow n_7 \rightarrow n_8 \rightarrow n_2$. (4) means that, if a branch condition is never satisfied at a jump statement, the branch will be removed, and the removed branch will not appear in the model $\tilde{M}$, such as $n_2 \rightarrow n_4$.

As mentioned in Section 3.1, circular reasoning must be avoided. Figure 8 is the CTL tree extended from the model of Figure 6. To validate the removal of the branch $3 \rightarrow 5$, the condition must have appeared previously, before the branch. In this example, the condition formula can only refer to nodes 1 and 2. Although the removal of a branch changes the structure of model, we only use the unprocessed part for validation. Therefore, the model without the removed branch will be correct.
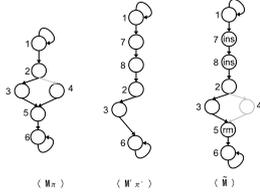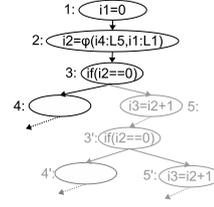

Fig. 7. Transition relation $\tilde{R}$

Fig. 8. CTL tree of program in Figure 6

Congruence set

Congruence [2][15] refers to different variables whose value is always equal under any execution of the program. There are several congruence-finding algorithms, such as those of Kildall [11], Alpern [2], and Rüthing [15]. Kildall's research analyzes the data flow and exactly computes the congruence set, but it requires computation of order $O(2^n)$. Alpern's method computes with order $O(n \, log(n))$, but it identifies all operators as different even though different operators sometimes give congruent computations. Rüthing's method is a combination of the Kildall's and Alpern's methods, but its order is $O(n^4 log(n))$. We use a unique algorithm that can compute almost with the accuracy of Kildall, but with order $O(n)$. Congruence sets are computed only for assignment instructions. Consider Figure 9(a), which is an example program, with 9(b) being its corresponding CTL tree. Figure 9(c) gives some results of congruence computation.

(i) For an assignment whose right-hand side is a constant or a variable substituted by a parameter:
  • Create a new congruence set if the right-hand side does not belong to any existing preceding node's congruence set. Here, we obtain $\{a0, 1\}$ for $a0 = 1$.
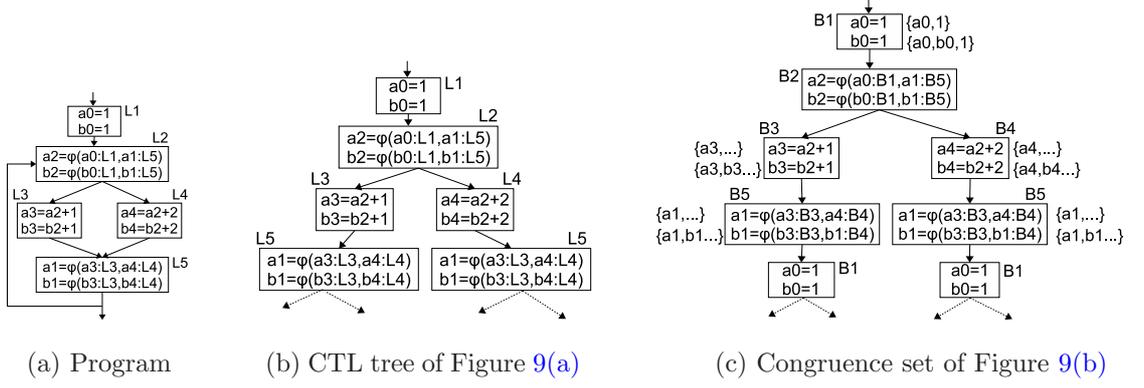
8

Fig. 9. Computation of congruence set

- If the right-hand side is in any existing congruence set, such as $b0 = 1$, a new congruence set is created with a new member being the left-hand side of the assignment in the existing one, namely $\{a0, b0, 1\}$. Let us name this set $cong1$.

(ii) For an assignment whose right-hand side is an expression such as $a2 = \phi(a0, a1)$ or $b2 = \phi(b0, b1)$, transform the expression as follows:

- Create a computation SSA Graph [2] whose parent is the left-hand of the object statement, as shown in Figure 10(a).
- Make the SSA Graph into a tree structure. Remove all the back edges and represent the node with a back edge of $ref : object\ node$ indicating what is referred to by the back edge. For example, the child node $ref : a2$ means it has a back edge that refers to the root of the tree. Figure 10(b) gives the results for Figure 10(a).

Fig. 10. SSA Graph and transformation

- Normalize the tree to a unified shape, as shown in Figure 10(c). (Sink the ϕ node and sort the child nodes into alphabetical order if it satisfies the condition to be unified. We use Rüthing's method to sink the ϕ node. The ϕ in a different basic block is different).
- Return the tree to the style of the formula. For both $a2$ and $b2$, we can get $\phi_2(cong1, +(\alpha, \phi_5(cong1, 2)))_\alpha$. The tiny-$\alpha$ subscript identifies the part bounded by parentheses before it, and this part will be extended again at large-α subscript. The formula symbolically expresses the computational behavior of a variable in a program. The variables $a2$ and $b2$ are congruent after the computation. After reforming the instruction, a new congruence set will be created following step (i).
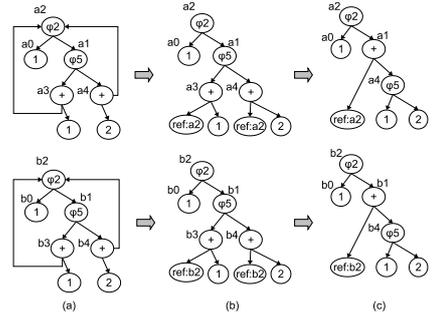
9

The variables $z1$ and $p3$ in the program of Figure 2 (right) can be judged as congruent by this method, but not by Alpern's method.

4.3 Define the semantically equivalent transformations and formalize them using CTL

We describe seven CTL formulas in Table 2 that describe the semantics of equivalence corresponding to the seven types of transformation of the program in Figure 11. $v, x, e \ldots$ are free variables that appear in CTL formulas. They must be bound to actual variables in an actual transformation.

name	transformation	CTL formula
$(a) rm\_def$	deletion of $v = x$	$\neg (E\ trans(v) U((use(v) \vee \neg rm\_use(v)) \vee ins\_use(v)))$
$(b) rm\_jump$	deletion of jump statement $L1 : jump(L0, L2)$	$\neg \overleftarrow{E}\ (true\ U\ (use(L1) \wedge \neg rpl(L1 \rightarrow L2))) \wedge$ $\neg E\ true\ U\ (use(L1) \wedge \neg rpl(L1 \rightarrow L0))$
$(c) rpl$	replacement of $v1 \rightarrow v2$	$\overleftarrow{A}\ (trans(v1) \wedge trans(v2))\ U\ equal(v1, v2)$
$(d) ins\_def$	insertion of $v = e$	$\neg ((\overleftarrow{E}\ trans(v)\ U\ def(v)) \wedge (E\ trans(v)\ U\ use(v)))$
$(e) ins\_jump$	insertion of jump statement $L1 : jump(L0, L2)$	$\neg \overleftarrow{E}\ (true\ U\ (use(L2) \wedge \neg rpl(L2 \rightarrow L1))) \wedge$ $\neg E\ (true\ U\ (use(L0) \wedge \neg rpl(L0 \rightarrow L1)))$
$(f) rm\_branch$	deletion of branch e if condition(e) is never satisfied	$\neg \overleftarrow{A}\ (true\ U\ condition(e))$
$(g) rpl\_edge$	movement of exception according to movement of e	false

<div align="center">

Table 2

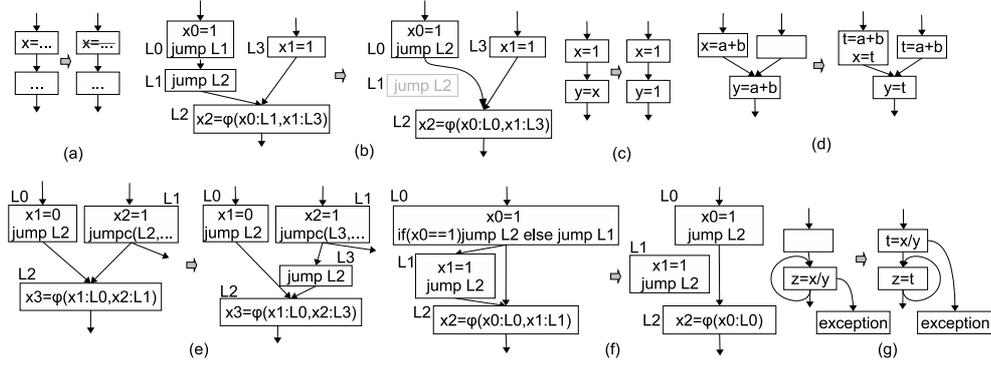Check formulas for optimization tranformations

</div>

Fig. 11. Transformations

SSA form will not change the control flow graph of a program, and the definitions of variables are unique. Therefore, all transformations of instructions and edges can be analyzed exactly for extraction without any help from the optimizer.

For example, if a variable is not used after it is defined, it is dead code and can be eliminated. We describe the semantics using formula rm_def in Table 2. It means that from the point of the deleted definition of the variable, no path exists that can reach a use of it without its being defined again. A use reached may be an instruction that uses the variable, and the instruction must not be deleted by optimization. Alternatively, a use may be an instruction

inserted by the optimization that also uses the variable. Such a path must not exist.

The dead code elimination here does not refer to the dead code elimination phase of optimizers, but the elimination transformation in all phases of optimizers. Therefore, the CTL formula describes the characteristic of transformation and does not rely on the optimization algorithm or its implementation. It has several advantages. First, the formulas do not need to be described individually for each optimization algorithm, and there are only a few formulas that can be used by all phases of optimizers. They can be adopted even when a new optimization is added or an optimization is modified. The user can formalize them very easily, without knowing either the optimization algorithm used in the system or its implementation.

4.4 Model checking [4] and bug reporting

At this stage, the model and check formulas are prepared. Every transformation is validated using the corresponding CTL formula via model checking. If any transformation does not hold, a report about the bug is given. The report contains information about the function name, line number, instruction, transformation, and the check formula.

We have three categories of bugs. In the first category are the semantic errors, and we report "*Fault*" for these bugs. These bugs are fatal and must be corrected. The second category contains bugs that will cause an error in some special cases, and we report "*Possible semantic error*" for them. The user should judge if it is necessary to correct these bugs, depending on the situation. The third category contains bugs that will not cause semantic errors, but that add redundancy to the optimized program. They are not fatal errors, but they should be fixed also. We report "*Redundancy alert*" for these bugs. Concrete examples will be given in Section 5.

5 Experiments

This section is about the experiments.

5.1 Optimizations adopted

In the backend of the COINS compiler [5], many optimizers that handle its Low-level Intermediate Representation (LIR) are implemented, in particular a rich set of optimizers [16] based on the SSA form. We applied our proposed method to the following optimizers, which operate on LIR [6]:

- Dead Code Elimination (DCE)
- Hoisting Loop-invariant Code (HLI)
- Copy Propagation (CPYP)

---

[6] " * " indicates an optimization that is newly validated by our method, compared to [17].

- Common Subexpression Elimination (CSE)
- Constant Folding and Propagation with Conditional Branches * (CSTP)
- Partial Redundancy Elimination with Question Propagation * (PREQP)
- Empty Block Elimination * (EBE)

Because our method does not depend on the algorithm, the verification can be carried out after each phase or after all phases of the optimization.

5.2   Environment

Our experimental data were acquired via the SPEC2000 benchmarks [19] on COINS 1.4.4.2, which is the newest version. There are about 6500 lines of code in our system (without the model checker).

The environment for the experiments is as follows. CPU: 750MHz Ultra-SPARCIII, OS: SunOS 5.8, JavaVM: 1.5.0 15, Heap Size: 512Mbyte, Memory: 1GByte.

5.3   Discovery of bugs

Using our method, several bugs were found. Two of them were semantic errors that have embarrassed the COINS development group for a long time. Several "*Possible semantic error*" and "*Redundancy alert*" bugs were also found. The bugs found by our method are shown in Figure 12, where (1)–(5) are unknown bugs, and (6) was found by previous work [17].
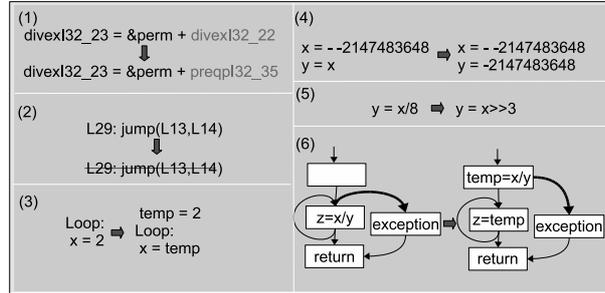
Fig. 12. Bugs

5.3.1   Semantic errors
Semantic error 1
The bug shown in Figure 12(1) was found when we applied CSE and PREQP to program 181.mcf. The fault report is as follows:

$FAULT : in\ sort\_basket.c$

$transformation : var\_rpl(divexI32\_22 \to preqpI32\_35)$

$at\ node\ 63 :\ divexI32\_23 = \&perm + divexI32\_22$

$CTL\ formula :$

$\overleftarrow{A}(trans(divexI32\_22) \wedge\ trans(preqpI32\_35)\ U\ equal(divexI32\_22, preqpI32\_35))$

This means that the replacement of variable "$divexI32\_22$" by "$preqpI32\_35$"

is not semantically equivalent. Following analysis, we discovered that the variable "$preqpI32\_44$" is necessary for the computation of "$preqpI32\_35$", but its definition cannot be found anywhere.

Semantic error 2

The bug shown in Figure 12(2) was found when applying CSTP and EBE to program 253.*perlbmk*, where an exception was thrown out. The fault report is as follows:

$FAULT : in\ md5.c$

$transformation : rm\_jump(L29 : jump(L13, L14))$

$at\ node\ 1605 : L29 : (jumpL14)$

$CTL\ formula :$

$\neg \overleftarrow{E}\ (true\ U\ (use(L29) \wedge \neg rpl(L29 \rightarrow L14))) \wedge \neg E\ true\ U\ (use(L29) \wedge \neg rpl(L29 \rightarrow L13))$

This means that "$L29$" is deleted, but it is still used somewhere. Following analysis, we discovered that the symbol representing the basic block's label "$L29$" is still referred to in the instruction $D.9\_20 = \phi(D.9\_19 : L29)$.

5.3.2 Possible semantic errors

Possible semantic error 1

This bug was found when we applied CPYP to some programs of SPEC CPU2000, as shown in Figure 12(4). If 2147483648 is an unsigned constant and - -2147483648 is a signed constant, they will have the same bit pattern, but substitution of an unsigned data value by a signed data item is implementation-dependent.

Possible semantic error 2

The bug shown in Figure 12(5) was found when we applied CSTP to some programs of SPEC CPU2000. Right-shift calculations are implementation-dependent. The result is correct if it is an arithmetic shift but incorrect if it is a logical shift with x being negative.

Possible semantic error 3

The bug shown in Figure 12(6) was found when we applied HLI to program 254.gap of SPEC CPU2000. It will cause an error only when y is 0. In fact, y is not 0.

5.3.3 Redundancy alert

The bug shown in Figure 12(3) was found when we applied HLI to any program of the SPEC CPU2000 benchmark. $x = 2$ is transformed to $x = temp$ and $temp = 2$ is inserted in the header of the loop. It will add redundancy to the program. Comparing the program optimized by HLI without and with the redundancy transformations, the execution time of the former for program 164.gzip of SPEC CPU2000 is improved by 27%.

5.4 Experiment on the efficiency of checking

In this section, we describe experiments that measure how much time was needed by the implemented checker in checking the optimizers. The experi-

ments were performed by measuring and comparing the compilation time with and without checking. Five measurements A, B, C, D, and E were made. A is the total compilation time without checking. B is the total compilation time when checking by our method and the validation is carried out after each phase of optimization. C is the total compilation time when the validation is carried out after all phases of optimizations. D is the number of checked transformations. E is the number of reports about bugs (but whether they accurately identify actual bugs is yet to be investigated).

The optimization used was the set of SSA optimizations performed when option $O2$[7] is specified, excluding OSR [6], which we cannot yet validate. The measurements are shown in Table 3. According to this table, the sum of compilation times for optimizations increases by a factor of 19.23 if we validate after each optimization phase. It is by no means fast, because our method validates a wider range of optimization, such as PREQP. The fact that our method works with higher accuracy is the essential feature. The sum of compilation times for optimizations increases by a factor of 7.27 if we validate after all the phases are performed. However, an incorrect transformation may disappear in another following phase, so we do not recommend this shortcut.

From the above experimental results, we can conclude that our method enables checking in a realistic time with high accuracy and is therefore reasonably practical.

	A	B	C	$\frac{B}{A}$	$\frac{C}{A}$	D	E	$\frac{E}{D}$
171.swim	15	173	46	11.53	3.07	2208	0	0
172.mgrid	13	359	80	11.97	2.67	3029	2	0.0007
179.art	17	418	94	19.00	4.27	1389	2	0.0014
188.ammp	254	10131	1078	30.70	3.27	3029	2	0.0007
175.vpr	191	16978	2390	63.12	8.88	12019	15	0.0021
181.mcf	53	334	116	5.39	1244	5.29	2	0.0016
197.parser	151	4669.00	1039	14.50	3.23	9416	26	0.0028
255.vortex	636	14660	6244	15.22	7.76	40724	36	0.0009
256.bzip2	29	1729	397	11.38	2.61	2997	9	0.0030
300.twolf	568	9335	2527	9.48	4.45	40254	3	0.0001
	sum(A)	sum(B)	sum(C)	$\frac{sum(B)}{sum(A)}$	$\frac{sum(C)}{sum(A)}$	sum(D)	sum(E)	$\frac{sum(E)}{sum(D)}$
sum	1927	58786	14011	19.23	7.27	116309	63	0.0005

Table 3

Comparison of times with and without verification (unit: second) and the number of transformations verified

6 Related work

There have been several pieces of research into optimization correctness. In Section 1, we categorized them into three kinds of approaches. Lacey et al. [12] and Lerner et al.'s [13] research guarantees theoretical strictness by defining a domain language that can be proven. However, these methods can deal only with simple optimizations, and they can only validate an optimizer that is written in the special domain language. In contrast, our method is not

---

[7] $O2 : -coins : hirOpt = cf$,
$ssa - opt = prun/divex/cse/cstp/hli/cstp/cpyp/preqp/cstp/dce/srd3, loopinversion$

limited by these conditions. The range of our validation includes the real world compiler that is overwhelmingly most often used.

Necula et al. [14] proposed a method that belongs to the same category (ii) as our work, and that checks the equivalence of the program semantics before and after optimization by symbolic inference and evaluation. But it uses a type of inference, and sometimes the inference fails. Moreover, it needs a little help from the optimizer. In contrast, because we operate on programs in SSA form, we can compare the program and extract all the differences easily and strictly. Therefore, our equivalence analysis can cover almost all transformations in the program (with the exception of type conversion). Moreover, our method does not need any help from the optimizer.

Sassa et al. [17], which is the same research group as that working on the verification of the COINS optimizer, proposed a method that proves the relations between annotations that correspond to the algorithm of the optimization. In contrast to the weakness of this work as mentioned in Section 1, our work does not rely on an algorithm, and users do not need to understand the algorithm of optimization or describe many check formulas for every optimization. Moreover, Sassa et al. indicated that CSTP had been verified, but we believe that the verification involves circular reasoning. In Figure 6, the condition for removal of branch $3 \rightarrow 5$ includes the deletion of statement $i3 = i2 + 1$. Therefore, in Figure 8, the object of validation has to be $3' \rightarrow 5'$ to trace $i3 = i2 + 1$. That is, the validation is to be done after node 3 itself has already been admitted.

7 Discussion and future work

7.1 Discussion

Originality of the work

First, our work does not rely on an algorithm. It heuristically validates the transformations with CTL, and with high accuracy. Second, it uses an SSA Graph to compute the congruence set. This is an original technique which expands the congruence set because our method can identify more congruent computation. Third, it can report redundancy.

Limitations of the SSA form

Although our method can extract simply all the differences between a program before and after optimization, it is limited to SSA form. However, SSA form is adopted by many compilers such as COINS, gcc [9], and many recent compilers. Therefore, it may be considered a rational choice.

Validation of transformations related to type

It was considered that the speed of the verification would slow greatly if transformations related to type were also objects of the analysis. Therefore, all types of constant were considered as the double type. This means that, if $0 \rightarrow 0.0$ happens, and it is an incorrect transformation, the bug will not

be reported. However, this would be very rare, and it did not happen in our experiments.

Conversely, a bug is reported sometimes even though it is an equivalent transformation, because the congruence set computation cannot recognize the type conversion. For example, the transformation $y = (double)x \rightarrow y = 0.0$ (x is previously assigned as $x = 0;$) will be reported as a bug. The number of such cases is very few, as shown in Table 3. Such a transformation is reported as *at node* $133 : \ divexF64\_12 = (double)divexF32\_16$, and it is possible to make these judgements by hand.

7.2 Future work

In the current system, several problems exist.

- The CTL formulas are defined according to intuition, and they must be proved.

- To be a complete validator, our method needs to validate all phases of the optimizer including OSR [6]. We think this can be done if a semantically equivalent transformation of an SSA graph by OSR can be defined.

- To improve the performance of the system, we need to eliminate reports that do not correspond to real bugs.

8 Conclusion

To our knowledge, our system is the first system that can validate realistic optimizers without any help from the optimizer or any need to enhance it. It does not rely on the algorithm, so there are only a few formulas required to check for almost all phases of the optimizer. It is also the first system that can check complex optimizations such as CSTP and PREQP. In addition, it can report on redundancy.

We found several different types of unknown bugs in COINS. Two of them were semantic errors that had embarrassed the COINS development group for a long time.

References

[1] Aho, A. V., Lam, M. S., Ravi, S. and Ullman, J. D.: Compilers Principles, Techniques, and Tools. Second Ed., Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] Alpern B., Wegman M. N., and Zadeck F. K.: Detecting equality of variables in programs, *In Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 1–11, 1988.

[3] Chiba, S., Nishizawa, M. and Kumahara, N.: GluonJ Home Page, http://www.csg.is.titech.ac.jp/projects/gluonj/

[4] Clarke, E. M. Jr., Grumberg, O. and Reled, D. A.: Model checking. The MIT Press, 1999.

[5] COINS Project: COINS Home Page. http://www.coins-project.org/

[6] Cooper, K. D., Simpson, L. T. and Vick, C. A.: Operator strength reduction, *ACM Trans. Prog. Lang. Syst.*, Vol. 23, No. 5, pp. 603–625, 2001.

[7] Cousot, P. and Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 238–252, Los Angeles, California. ACM Press, New York, 1977.

[8] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 4, pp. 451–490, 1991.

[9] gcc Home Page. http://www.gnu.org/software/gcc

[10] Jaramillo, C., Gupta, R. and Soffa, M. L.: Debugging and testing optimizers through comparison checking, *Electronic Notes in Theoretical Computer Sciences*, Vol. 65, No. 2, pp. 1–17, 2002.

[11] Kildall, G. A.: A unified approach to global program optimization, *1st ACM Symposium on Principles of Programming Language,* pp. 194–206, October 1973.

[12] Lacey, D., Jones, N. D., Van Wyk, E. and Frederiksen, C. C.: Proving correctness of compiler optimizations by temporal logic. *Proceedings of Symposium on Principles of Programming Languages*, pp. 283–294, 2002.

[13] Lerner, S., Millstein, T., Rice, E. and Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 364–377, 2005.

[14] Necula, G. C.: Translation validation for an optimizing compiler, *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 83–94, 2000.

[15] Rüthing, O., Knoop, J. and Steffen, B.: Detecting equalities of variables in program: combining efficiency with precision, *Proc. 6th Int. Static Analysis Symposium (SAS'99), Lecture Notes in Computer Science 1694*, Vol. 1694, pp. 232–247, 1999.

[16] Sassa Laboratory: Optimization in Static Single Assignment Form-External Specification, 2007. http://www.is.titech.ac.jp/~sassa/coins-www-ssa /english/ssa-external-english.pdf

[17] Sassa, M. and Sahara, S.: Validating correctness of compiler optimizer execution using temporal logic. *Seventh International Workshop on Compiler Optimization meets Compiler Verification (COCV 2008)*, pp. 1–17, April 2008.

[18] Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations, *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM New York, NY, USA, pp. 38–48, 1998.

[19] SPEC.: SPEC Home Page. http://www.spec.org/

[20] Takimoto, M. and Harada, K. Efficient question propagation, in [16].

[21] Van Leeuwen, J. (Ed.): Handbook of Theoretical Computer Science Vol. B, Formal Models and Semantics, Elsevier Science Publishers B. V., 1990.

[22] Wegman, M. N. and Zadeck, F. K.: Constant propagation with conditional branches, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 2, pp. 181–210, 1991.