

Validating Correctness of Compiler Optimizer Execution Using Temporal Logic

Masataka Sassa^{1,2} Soichiro Sahara³

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

Abstract

It is very important that compiler optimization works correctly without changing the semantics of a program. However, because there are many complex optimizations, it is generally difficult to implement them correctly. In this paper, we propose a technique for validating whether or not the optimization transformations to the program have been performed correctly, *after* the execution of the optimizer. We first describe the properties that program points modified by the optimization have to satisfy to preserve the program semantics, in terms of temporal logic. The system then performs model checking on the optimized program, to check if these program points satisfy the logical formulas described. This technique has the advantages that it can be applied to complex optimizers that already exist, and that checking occurs within a realistic time. We have implemented and executed this technique and found an unknown bug in an optimizer within a widely-used compiler.

Keywords: compiler optimization, validation, model checking, CTL

1 Introduction

1.1 Background

Optimization in compilers is an important technique and is being actively investigated. However, optimizations are often complex, and bugs that change the program semantics may easily be introduced in several phases, including algorithm design and implementation. Moreover, bugs in optimizations are generally difficult to find and to remove, for the following reasons:

- Even if the optimization seems to be achieved normally, the optimized object code may cause unintended behavior. This cannot be discovered until the object code is run, and sometimes not even then.
- When we find a bug that changes the meaning of the object code, it is difficult to identify which part of the optimizer created the erroneous transformation.

¹ An earlier version of parts of this article appeared in the “Computer Software” journal published by the JSSST (in Japanese).

² Email: sassa@is.titech.ac.jp

³ Email: soichiro.sahara@gmail.com

If bugs exist in optimizers, the program compiled with optimization is not guaranteed to behave correctly. This background shows that techniques for assuring that there are no bugs in optimizers are quite important. Even if it is not possible to completely assure that there are no bugs in *optimizers*, *at least* we must guarantee that the semantics of the *optimized program* is not changed.

Previous work that improves the reliability of compiler optimizers includes:

- Validating that the optimizers themselves are correct. Validated optimizers can optimize any program without changing its behavior.
- Verifying by checking that the transformation did not change the semantics of the program after executing the optimizers. Checked programs can be inferred to be correctly optimized.

Validation studies include Lacey et al.’s work [9] and Lerner et al.’s work [11]. However, they cannot deal with complex optimizations. Verification studies include Rinard and Marinov’s work [13] and Necula’s work [12]. Rinard and Marinov’s work can show the strict correctness of program transformation, but it is not clear how practical it is if applied in a real setting. The work of Necula is, by contrast, quite practical, but there is no guarantee of strict correctness.

1.2 Outline

In this paper, we propose a method that checks whether program transformation by the optimizer preserves the behavior of the program *after* the execution of the optimization.

We use the temporal logic formula CTL-FV [9]. We first describe, in terms of CTL-FV, a property that each part of the program changed by optimization transformation must satisfy to preserve the program semantics. Then we check whether the property is satisfied, by model checking *after* execution of the optimization. If all checks succeed, the optimization has executed correctly, and we can conclude that the program semantics is preserved.

The advantages of our method are as follows:

- It can be applied to existing optimizers.
- It can check a broader range of optimizers than those that can be handled by Lacey et al.
- Identification of cause is easy when bugs are found.
- Checking occurs within a practicable time.

In order to confirm the applicability of our method, we applied this method to several optimizers implemented in the widely-used COINS (COmpiler INfraStructure) compiler [5]. As a result, we found an unknown bug in the optimizer for loop invariant code motion.

2 Program Optimization

Program optimization is a program transformation made by the compiler for the purpose of improving the execution speed or the compactness of the program. The optimization is generally made by *analyzing* the characteristics of the program, and *transforming* it based on these results.

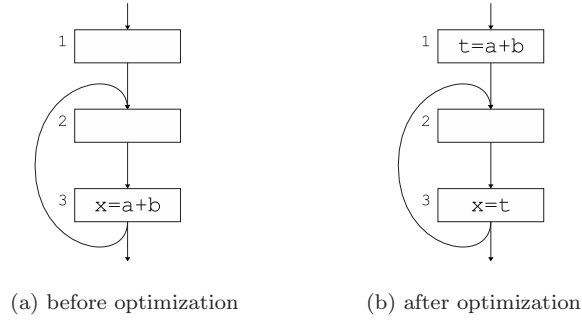


Fig. 1. Example of loop invariant code motion

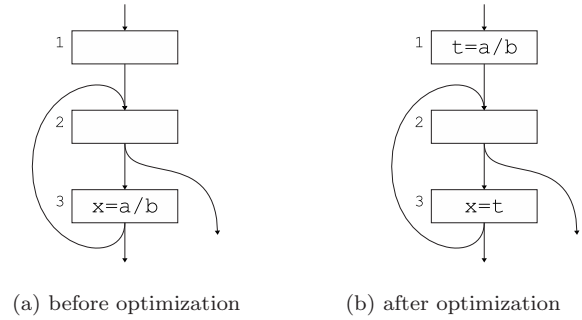


Fig. 2. Example of incorrect optimization of loop invariant code motion

2.1 Correctness of optimization

What is required, in optimization, is at least to have the same program behavior before and after optimization. When the behavior of the program does not change, we say that *the program semantics is preserved*. Optimization that does not preserve the program semantics is an incorrect optimization.

Apart from the correctness of optimization, an optimization is required to raise the efficiency of the program. However,

- There are cases where a combination of optimizations gives a better result than applying each of them individually.
- Usually, an optimizer improves only *conservatively* if no profile information is given.

Therefore, whether or not efficiency is really improved by an optimizer is a difficult problem and cannot be shown in general. From now on, when we say “optimization is executed correctly”, this means that at least the optimized program preserves its semantics.

2.2 Example of optimization and its correctness

2.2.1 Loop invariant code motion

An expression whose value is always the same during a loop is called a *loop invariant expression*. An example of a loop invariant expression is an expression whose operands are all variables defined outside the loop or are constants. A loop invariant expression has the same value even if its computation is made outside the loop. Therefore an optimization to compute its value before entering the loop reduces the number of computations at runtime. This optimization is called *loop invariant code*

motion.

Fig. 1 is an example of loop invariant code motion. In Fig. 1(a), because $a + b$ is an expression whose value does not change inside the loop, we compute $a + b$ before the loop and assign its value to a temporary t as in Fig. 1(b). Use of the original $a + b$ in the loop is replaced by the temporary t . Such a move of the point of computation of an expression from the original point to a point before the loop is called *hoisting an expression*.

2.2.2 Correctness of loop invariant code motion

In the example of loop invariant code motion in Fig. 1, modification for optimization takes place at the following two points:

- hoisting $a + b$, that is, inserting $a + b$ before the loop
- use of the original $a + b$ is replaced by a temporary t .

We call these the *insertion point* and *replacement point*, respectively.

Below, we explain the property that each transformation must satisfy, so that loop invariant code motion does not change the program semantics.

Insertion of hoisted expression

In order that insertion of $t = a + b$ does not change the program semantics, the following condition must hold:

- the value of t defined at inserted statement $t = a + b$ is not used at points other than the replacement point.

If the expression to be hoisted is an expression that might cause an exception such as a/b , further caution is necessary. Fig. 2 is an example of hoisting a/b , but it is incorrect for the following reason. In Fig. 2(a), there may be an execution path that does not go through node 3. Therefore, an execution such as the following exists: “Although the value of b is 0, the execution exits the loop without passing node 3, causing no exception.” However, if we hoist a/b as in Fig. 2(b), a/b is always computed, and it may cause an exception that did not occur originally. Therefore, hoisting and inserting an expression that may cause an exception must further satisfy the following condition:

- We do not insert a computation in an execution path that did not originally contain the computation.

Replacement of use of expression

For the replacement of $a + b$ by t to preserve the program semantics, the following condition must be satisfied:

- the values of $a + b$ and t are equal.

In other words, if there are no statements that define t , a and b between the insertion point and the replacement point, the program semantics does not change.

3 Temporal Logic

Our method uses model checking based on the temporal logic CTL-FV. In this section, we first describe the model of the program that is subject to being checked, and then present the syntax and semantics of CTL-FV.

$$L(n) =$$

$\{$	$node(N)$	$ $	$n = N$ ⁵	$\}$	
\cup	$\{$	$block(B)$	$ $	n is a statement in basic block B	$\}$
\cup	$\{$	$use(X)$	$ $	variable X is used in n	$\}$
\cup	$\{$	$def(X)$	$ $	variable X is defined in n	$\}$
\cup	$\{$	$comp(E)$	$ $	expression E is computed in n	$\}$
\cup	$\{$	$trans(E)$	$ $	expression E is not changed in n , that is, variables in E are not defined in n	$\}$
\cup	$\{$	$mark(M)$	$ $	n has a mark M	$\}$

Table 1
Definition of $L(n)$

3.1 Model of the program

In order to perform model checking by CTL-FV, we must formally represent the program as a state transition model. As the optimization of a program is performed on a control flow graph, it is natural for the state transition model used in our method to be based on a control flow graph [1].

Definition 3.1 (*control flow model*) We consider a control flow graph $G = (N, E)$ where each node corresponds to a statement. Here, N is a set of nodes and E is a set of directed edges between nodes. The set of all atomic predicates is denoted by AP . For each node $n \in N$, we denote the mapping which gives the set of atomic predicates $L(n)$ that hold at that node by the mapping $L: N \rightarrow 2^{AP}$. The triplet $M = (N, E, L)$ is called the *control flow model*. This is a Kripke structure [4], where N corresponds to the set of *states*, and $E \subseteq N \times N$ corresponds to the *transition* between states. If there is a transition between states n and n' , i.e., $(n, n') \in E$, it is often denoted by $n \rightarrow n'$. From now on, the term *model* refers to this control flow model. ⁴

The definition of $L(n)$ in a control flow model $M = (N, E, L)$ is shown in Table 1.

Definition 3.2 (*path on a control flow model*) In a control flow model $M = (N, E, L)$, if an infinite sequence of states n_0, n_1, n_2, \dots satisfies $n_i \rightarrow n_{i+1}$ for any $i \geq 0$, this infinite sequence is called an *infinite path*. If a finite sequence of states n_0, n_1, \dots, n_m satisfies $n_i \rightarrow n_{i+1}$ for any i ($0 \leq i < m$), and $\forall n \in N, \neg(n_m \rightarrow n)$, this finite sequence is called a *finite path*. An infinite path and a finite path together are called a *path*.

When $n_1 \rightarrow n_2$ holds, the reverse of this transition relation is called a *reverse transition*, and it is denoted by $n_2 \rightarrow^\circ n_1$. The path defined for this reverse transition is called a *reverse path*.

⁴ We do not put infinite loops at the entry node and the exit node.

⁵ What is attached to node 1 is $node(1)$.

$\phi ::= true$	$\phi ::= E \psi$	$\psi ::= X \phi$	$\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$
$\phi ::= false$	$\phi ::= A \psi$	$\psi ::= \phi U \phi$	$\phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$
$\phi ::= \alpha$	$\phi ::= \overleftarrow{E} \psi$	$\psi ::= \phi W \phi$	$EF \phi \equiv E(true U \phi)$
$\phi ::= \neg\phi$	$\phi ::= \overleftarrow{A} \psi$		$AF \phi \equiv A(true U \phi)$
$\phi ::= \phi \wedge \phi$			$\overleftarrow{EF} \phi \equiv \overleftarrow{E}(true U \phi)$
			$\overleftarrow{AF} \phi \equiv \overleftarrow{A}(true U \phi)$

Table 2
Syntax (left) and Syntax sugar (right) of CTL-FV

3.2 CTL-FV

The temporal logic used in our method is *CTL-FV* [9] proposed by Lacey et al. It is a logic based on CTL [3,4], which is a type of branching-time temporal logic, and has the following distinctive features:

- It can use quantifiers \overleftarrow{E} and \overleftarrow{A} , which can deal with reverse paths.
- Atomic propositions are generalized to predicates that can use free variables as parameters.

Intuitively, \overleftarrow{E} and \overleftarrow{A} are path quantifiers made by just reversing the direction of the usual path quantifiers E and A , respectively.

There are three reasons for our adoption of CTL-FV from among the many temporal logics:

- It is a logic following the execution paths, and is easy to understand intuitively.
- It can handle the reverse paths naturally, and is suited to describing the characteristics of control flow and dataflow.
- It is possible to perform model checking efficiently.

Syntax of CTL-FV

The syntax of CTL-FV is shown in Table 2 (left). ϕ is a nonterminal deriving expression concerning states (*state expressions*), ψ is a nonterminal deriving expression concerning paths (*path expressions*), and α is a predicate with free variables as parameters.

Combinators that are often used, but which do not appear in the syntax rules, are defined as abbreviations, as shown in Table 2 (right).

Semantics of CTL-FV

We write $M, n \models \phi$ when state expression ϕ holds at state n on model M . We write $M, p \models \psi$ when path expression ψ holds on path p . In both cases, we often omit M when M is understood, and simply write $n \models \phi$ and $p \models \psi$, respectively. The definition of the semantics of CTL-FV for the control flow model is shown in Table 3.

state formulas

$n \models true$	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 $n \models \phi_2$
$n \models E\psi$	iff	$\exists p (n = n_0 \rightarrow n_1 \dots): (n_i)_{i \geq 0} \models \psi$
$n \models A\psi$	iff	$\forall p (n = n_0 \rightarrow n_1 \dots): (n_i)_{i \geq 0} \models \psi$
$n \models \overleftarrow{E}\psi$	iff	$\exists p (n = n_0 \rightarrow^\circ n_1 \dots): (n_i)_{i \geq 0} \models \psi$
$n \models \overleftarrow{A}\psi$	iff	$\forall p (n = n_0 \rightarrow^\circ n_1 \dots): (n_i)_{i \geq 0} \models \psi$

path formulas ($p = n_0 \rightarrow' n_1 \dots$, \rightarrow' is \rightarrow or \rightarrow°)

$p \models X\phi$	iff	n_1 exists and $n_1 \models \phi$
$p \models \phi_1 U \phi_2$	iff	$\exists i \geq 0 [n_i \models \phi_2$ and $\forall j [0 \leq j < i$ implies $n_j \models \phi_1]]$
$p \models \phi_1 W \phi_2$	iff	$(p \models \phi_1 U \phi_2)$ or $(\forall k \geq 0 [n_k \models \phi_1$ and n_{k+1} exists])

Table 3
Semantic definition of CTL-FV

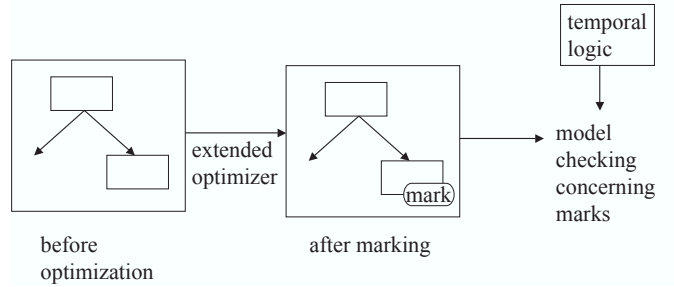


Fig. 3. Outline of the proposed method

4 Proposed Method

4.1 Outline of the proposed method

In this paper, as outlined in section 1.2, we propose a method that checks whether the program transformation made by the optimizer has preserved the program semantics *after the execution of optimization*. Fig. 3 shows the outline of the proposed method. The check is made as follows:

- We call the points of the subject program transformed by the optimizer *transformed points*. For each transformed point, use the temporal logic CTL-FV to

describe, beforehand, the property of the point that must hold to preserve the program semantics.

- Check, by model checking, whether all transformed points satisfy the described formulas after the execution of optimization.
- If all checks succeed, we judge that the optimization was executed correctly. If any check fails, the corresponding transformation is erroneous, and we judge that there are bugs in the optimizer.

To check the transformed points of the subject program after execution of the optimizer, we put *marks* corresponding to the type of transformation at the transformed points during optimization. We can include the marking by extending the optimizers.

We will now introduce some terminology. We denote the parts of the optimizer that perform transformations to the subject program for the purpose of optimization as *transformation points*. In addition, we also denote the transformation points of the optimizer extended for adding marks to the subject program as *extension points*. Therefore, we introduce code to add marks to the transformed points of the subject program at the extension points of the optimizer. This extension can be made easily, with almost no modification to the source code of the optimizer itself, by using *aspect-oriented programming*.

Our method does not validate the correctness of the optimizer itself, but it is a method that checks whether the *result* of the execution of the optimizer is correct.

4.2 Steps to realization of the proposed method

The proposed method comprises the following five steps:

■ Preparation:

- (i) For each optimization, describe the condition for the correctness of the optimization, as a *specification* to be checked.
- (ii) Extend the existing optimizer so that marks can be added to transformed points of the subject program during optimization.

■ Before optimization:

- (iii) Make the model of the program.

■ During the execution of the optimization:

- (iv) Mark the points of the subject program transformed by the optimizer.

■ After the optimization:

- (v) Perform model checking.

The details of each step will be explained in the following. We will use the optimization of loop invariant code motion shown in section 2.2 as an example during the explanation. The points that are transformed in loop invariant code motion were the insertion point and the replacement point. Fig. 4(b) is the flow graph after the optimization, where marks are added at each transformed point. The insertion point is marked by $(ins, t, a + b)$, and the replacement point is marked by $(rpl, t, a + b)$, showing the type of transformation at each point.

Step1: Description of the checking specification

In our method, we describe in advance, using CTL-FV, the specification to be checked, which the transformed points of the optimization must satisfy to preserve

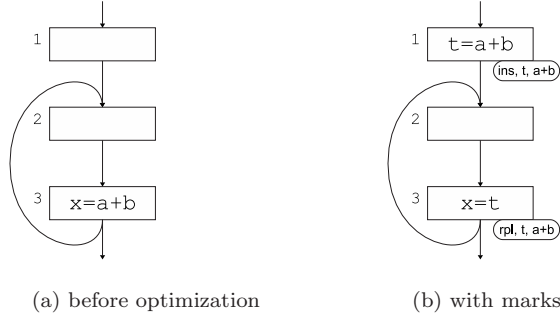


Fig. 4. Example of loop invariant code motion (with marks)

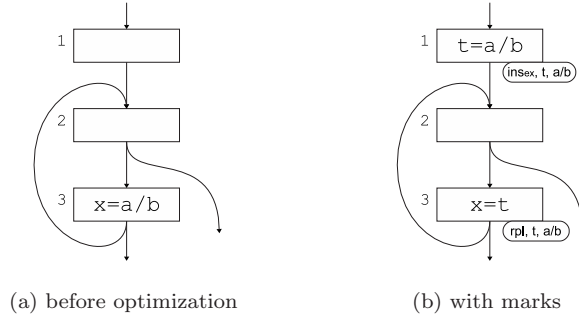


Fig. 5. An example of erroneous loop invariant code motion (with marks)

the program semantics. We call this description the *checking specification*, or simply the *specification*. Because the method of transformation and the property that the transformation should satisfy will differ, depending on the optimization, the description must be individually described for each optimization.

As presented in section 4.1, each transformed point of the subject program requires its own way of marking, and so each is marked individually. Therefore, it is natural that a description is given for each type of mark. If we denote a mark by m and the corresponding CTL-FV formula by ϕ , the specification is expressed as their pair $\langle m, \phi \rangle$.

The point where the hoisted expression is inserted

As presented in section 2.2, correctness at the point where $t = a + b$ is inserted requires that:

- The value of t , defined at the inserted statement $t = a + b$, is not used at points other than the replacement point.

In other words,

- There are no paths starting from the insertion point such that “ t is used at points other than the replacement point without redefinition of t ”.

If we denote t by the free variable t , expression $a + b$ by the free variable e , and consider that the replacement point is marked by (rpl, t, e) , the CTL-FV formula expressing this is:

$$\neg E (\neg def(t) U (use(t) \wedge \neg mark(rpl, t, e))) \tag{1}$$

Therefore, if we denote formula (1) by ϕ_1 , the specification to be satisfied is $\langle (ins, t, e), \phi_1 \rangle$.

In cases where an expression which may cause an exception, such as a/b , is hoisted and inserted, the following condition is also necessary:

- Do not insert a computation of a/b in the execution path where there was originally no computation of a/b .

In order to distinguish this from the hoisting of expressions that do not cause exceptions, the point where an exception-prone expression such as division is hoisted and inserted is marked by $(\mathbf{ins}_{\mathbf{ex}}, t, e)$, as in Fig. 5(b). We can rephrase the above as:

- In all paths starting from the insertion point with this mark, we reach the replacement point.

If we denote a/b by a free variable e , the CTL-FV formula expressing this is⁶:

$$A(\text{true } W \text{ mark}(\mathbf{rpl}, t, e)) \quad (2)$$

Therefore, if we denote formula (1) by ϕ_1 and formula (2) by ϕ_2 , the specification to be satisfied is $\langle (\mathbf{ins}_{\mathbf{ex}}, t, e), \phi_1 \wedge \phi_2 \rangle$.

The point where use of an expression is replaced

As presented in section 2.2, correctness at the point where use of $a + b$ is replaced by \mathbf{t} requires that:

- The values of $a + b$ and \mathbf{t} at the replacement point are equal.

In other words,

- In all reverse paths starting from the replacement point, the values of $a + b$ and \mathbf{t} do not change until they reach the insertion point.

Unlike the case of the insertion point, this same condition is sufficient for replacement of an expression that may cause an exception, such as a/b .

If we denote a temporary variable \mathbf{t} by a free variable t and an expression $a + b$ by a free variable e , the replacement point is the node marked by either the mark (\mathbf{ins}, t, e) or the mark $(\mathbf{ins}_{\mathbf{ex}}, t, e)$. Therefore, the CTL-FV formula is⁶:

$$\overleftarrow{A} ((\neg \text{def}(t) \wedge \text{trans}(e)) \ W \ (\text{mark}(\mathbf{ins}, t, e) \vee \text{mark}(\mathbf{ins}_{\mathbf{ex}}, \mathbf{t}, e))) \quad (3)$$

Therefore, if we denote formula (3) by ϕ_3 , the specification to be satisfied is $\langle (\mathbf{rpl}, t, e), \phi_3 \rangle$.

Step2: Extension of the existing optimizer

In our method, in order to realize the marking to the *transformed point* of the subject program during the execution of the optimizer, we need to insert code for adding marks into those sections of the *transformation points* of the optimizer that perform program transformation.

This insertion of code into the optimizer can utilize aspect-oriented programming, which can minimize the direct modification of the source code in the existing optimizer. The only modification of the optimizer source code is the reluctant insertion of the method call with an empty statement, for specifying the code insertion point. We implemented this using the Java aspect-oriented system GluonJ [2].

Step3: Modeling of the program

⁶ The reason for using a W operator instead of a U operator is to avoid having to consider paths that do not reach the end node, such as infinite loops.

In the proposed method, we model the control flow graph as a control flow model, as presented in section 3.1. With respect to the model, we could consider utilizing the control flow graph as is, which is an intermediate form of the compiler. However, this approach has the following problem:

- In an optimization such as dead code elimination, which deletes statements, if we delete a node of the graph itself, we cannot identify the deleted node in the post-optimization flow graph. If we cannot identify the point of deletion, we cannot check the transformed point.

The simplest and most robust solution to this issue is to make a copy of the control flow graph before optimization, and use it as the model. Each time the optimizer performs a transformation on the original program, we make a matching modification to the model copy. Then, for a transformation that deletes a node, we cope with it by changing the node copy to “skip”.

Our implementation uses this method to handle the issue of node deletion.

Step4: Marking the transformed points

During optimization, we add marks to the program according to the transformation, as shown in Fig. 4(b). This occurs automatically because we have extended the optimizer.

Step5: Model checking

After execution of the optimization, the model of the subject program to be checked and the set of checking specifications are obtained. Using them, we check whether the transformed points of the program satisfy the property that preserves the semantics.

Because specification $\langle m, \phi \rangle$ means

- ϕ should hold at the node marked by m ,

this has the same meaning as the following formula:

$$\forall n \in N, n \models \text{mark}(m) \rightarrow \phi \quad (4)$$

The model checking is done by translating the specification into a single CTL-FV formula in the form of formula (4).

Incidentally, a CTL-FV formula containing free variables cannot be model-checked as is. We have to bind free variables in the CTL-FV formula to symbols actually appearing in the program, and transform it into a formula containing no free variables. This binding is performed by utilizing the marks added during the optimization.

In the example of Fig. 4(b), we added two marks. The CTL-FV formula corresponding to mark $(\text{ins}, \mathbf{t}, \mathbf{a} + \mathbf{b})$ is formula (1). If we consider the correspondence between the actual mark and formula (1), \mathbf{t} corresponds to t and $\mathbf{a} + \mathbf{b}$ corresponds to e . Therefore, we bind formula (1) using them. This gives formula (5):

$$\neg E (\neg \text{def}(\mathbf{t}) U (\text{use}(\mathbf{t}) \wedge \neg \text{mark}(\text{rpl}, \mathbf{t}, \mathbf{a} + \mathbf{b}))) \quad (5)$$

Similarly, if we bind formula (3), which corresponds to $(\text{rpl}, \mathbf{t}, \mathbf{a} + \mathbf{b})$, we get formula (6):

$$\overleftarrow{A} ((\neg \text{def}(\mathbf{t}) \wedge \text{trans}(\mathbf{a} + \mathbf{b})) W (\text{mark}(\text{ins}, \mathbf{t}, \mathbf{a} + \mathbf{b}) \vee \text{mark}(\text{ins}_{\text{ex}}, \mathbf{t}, \mathbf{a} + \mathbf{b}))) \quad (6)$$

From these two formulas and formula (4), the final formulas to be used in the model checking are:

$$\models \text{mark}(\mathbf{ins}, \mathbf{t}, \mathbf{a} + \mathbf{b}) \rightarrow \phi'_1 \quad (7)$$

$$\models \text{mark}(\mathbf{rpl}, \mathbf{t}, \mathbf{a} + \mathbf{b}) \rightarrow \phi'_3 \quad (8)$$

Here, ϕ'_1 is formula (5), and ϕ'_3 is formula (6).

There are several ways of performing model checking. Our implementation used the algorithm based on classical CTL model checking given in [3].

5 Application to Actual Optimizers

In the backend of the COINS compiler [5], many optimizers that handle its Low-level Intermediate Representation (LIR) are implemented, especially a rich set of optimizers based on the Static Single Assignment (SSA) form.

We applied our proposed method to the following optimizers which operate on LIR:

- In the SSA form: loop invariant code motion, constant propagation with conditional branches, copy propagation, common subexpression elimination, dead code elimination.
- In the normal (non-SSA) form: lazy code motion [8]

Due to space limitation, the details are omitted. Application to loop invariant code motion and constant propagation with conditional branches are given in [14].

6 Experiments

In this section, we describe experiments that apply the proposed method to the optimizers in COINS, and consider the usefulness of our method based on the experimental results.

6.1 Discovery of an unknown bug

We found an unknown bug by checking, via our method, the loop invariant code motion optimizer in SSA form implemented in COINS. This is a bug that causes erroneous hoisting of an expression that may cause an exception such as that in Fig. 5.

This bug was found when compiling and checking the program 254.gap of the SPEC CPU2000 benchmark. This program generated object code that ran normally even when we performed SSA optimizations of COINS. Actual execution of this object code does not cause division by 0, and so no bugs are found this way. However, by using our method, the latent bug is found. This is a remarkable feature of our method.

The bug was found as follows:

- (i) During checking, a counterexample was detected in the model checking of the specification corresponding to $\text{mark}(\mathbf{ins}_{\text{ex}}, t, e)$. This specification is the one for checking if an expression can be hoisted.

Optimizer	Lines	Extension	Rewriting
Loop invariant code motion	357	2	0
Constant propagation with conditional branches	1143	5	2
Copy propagation	143	3	2
Common subexpression elimination	498	7	1
Dead code elimination	480	3	0
Lazy code motion	1259	2	0

Table 4
Number of extension points for each optimizer

- (ii) When we referred to the corresponding point in the source code of the optimizer, it gave no consideration to the case of an expression that could cause an exception.

In our method, the fact that model checking detects a counterexample means that there was an error in the transformed point corresponding to the specification being checked. In other words, our method has the characteristic features that we can directly find which transformation in optimization contains a bug from a counterexample, and that identification of the cause is easy after a bug is found.

6.2 Number of extension points of the optimizers

In our method, we have to extend the optimizer so that the transformed points of the subject program changed by the optimizer can be marked. In general, since the transformation points of the optimizers are few, this extension requires little effort. Furthermore, by utilizing aspect-oriented programming, there are quite a few points where it is actually necessary to rewrite the source code of the optimizer.

Table 4 shows the number of extension points for each optimizer. The columns refer to the following. Lines: number of lines of the source code of the optimizer, Extension: number of extension points of the optimizer, Rewriting: number of lines of source code rewritten in the optimizer.

From Table 4, we see that the number of extension points for adding marks was quite small compared to the number of lines in the optimizer. Moreover, the extension points, which correspond to the points where program transformation is performed, are often distinctive⁷. Therefore, it is easy for authors who have a general knowledge of optimizers to find them.

Note that no false alarms are reported.

6.3 Cost of describing the checking specification

Amount of checking specification

A checking specification must be described for each kind of mark. The number of types of marks corresponds to the number of extension points in Table 4. As shown above, there are few extension points. Moreover, a checking specification can usually be described in one to three lines, with a maximum of about five lines. From the above, we can say that the quantity of description in the specification is small.

Ease of description

⁷ For example, method calls that operate on the control flow graph.

To describe the checking specification in our method, it is necessary to accurately understand the algorithm and characteristics of each optimizer. Therefore, not everyone can easily describe it. If users are not optimizer writers, they must understand how the optimizer behaves, by reading the specification of the optimizer, or the research papers about the optimizer, or its source code.

In our case, writing the checking specification of the six optimizers in Table 4 required about three weeks, including the time to understand the precise behavior of the optimizers from a reading of the source code.

6.4 Experiments on efficiency of checking

As presented in section 4.1, our method performs a check each time an optimization is executed. Therefore, it is necessary that the checking time is within realistic limits. In this section, we describe experiments which measure how much time was needed by the checker implemented in the optimizers.

The environment for the experiments is as follows. CPU: Intel Pentium 4 CPU 2.80GHz, OS: Linux 2.6.17-13msmp, JVM: 1.5.0_08, Heap Size: 256Mbyte, Stack Size: 2048Kbyte, COINS: 1.4.1, GluonJ 1.2 (Aspect-oriented system), Benchmark: SPEC CPU 2000 version 1.2.

The experiments were performed by measuring and comparing the compilation time with and without checking. There were four items measured:

- A The sum of times needed for executing optimizations without checking.
- B The sum of times needed for executing optimizations with checking by our method.
- C Total compilation time without checking.
- D Total compilation time with checking by our method.

The optimization used was the set of SSA optimizations performed when option O2 is specified.⁸

The measurements are shown in Table 5. According to this table, the sum of execution times for optimizations increases by a factor of 15.08, and the sum of total compilation time increased by a factor of 1.67, when checking is performed.⁹ The times of optimization and compilation with checking are by no means fast. However, considering that the increase of compilation time by adding checking is from 27 minutes (without checking) to 43 minutes (maximum time with checking) for 177.mesa, we think this is a realistic time.

We consider that the checking in our method could be made more efficient by tuning our implementation.

From the above experimental results, we can conclude that our method can be checked in realistic time and is therefore reasonably practical.

⁸ Lazy code motion is not included because it was only at the prototype stage.

⁹ The reasons why $B - A = D - C$ does not hold seem to include the influence of the time of weaving by GluonJ, the timing of JIT compilation, and the execution time for garbage collection.

	A	B	$\frac{B}{A}$	C	D	$\frac{D}{C}$
175.vpr	6.85	43.56	6.35	325.79	487.49	1.49
181.mcf	1.78	8.99	5.05	47.86	98.45	2.05
186.crafty	14.21	380.29	26.74	303.51	834.97	2.75
197.parser	5.92	29.49	4.97	364.93	504.66	1.38
254.gap	27.17	400.42	14.73	1541.74	2303.60	1.49
255.vortex	21.32	112.57	5.27	1175.71	1675.16	1.42
256.bzip2	1.25	11.74	9.34	108.85	149.58	1.37
300.twolf	25.03	475.61	19.00	459.80	1234.32	2.68
171.swim	0.52	12.46	23.70	10.21	27.59	2.70
172.mgrid	0.81	17.10	21.02	19.12	42.81	2.23
177.mesa	29.53	540.70	18.30	1654.83	2622.32	1.58
179.art	0.53	3.66	6.89	30.43	43.41	1.42
183.equake	1.00	23.69	23.68	50.63	88.18	1.74
188.ammp	9.88	140.19	14.18	281.46	554.08	1.96
	sum(A)	sum(B)	$\frac{\text{sum}(B)}{\text{sum}(A)}$	sum(C)	sum(D)	$\frac{\text{sum}(D)}{\text{sum}(C)}$
sum	145.86	2200.52	15.08	6374.93	10666.66	1.67

Table 5
Comparison of compilation time with and without checking (unit: second).

7 Related Work

There has been much research concerning optimization correctness. This can be categorized into two approaches.

The first approach validates that the optimizers themselves are correct. Lacey et al. proposed a method that describes the dataflow analysis of optimizers using the temporal logic CTL-FV, and that executes the optimizer by model checking [9]. There are limitations to the optimizer, such as conditional constant propagation that cannot be handled. Lerner et al. proposed a system that describes the optimizer using an original domain-specific language based on temporal logic, and executes the optimizer [10] [11]. It has the feature of being able to perform the proof of the correctness of the optimizer almost automatically, using a theorem prover.

The second approach verifies by checking that the transformation did not change the semantics of the program after executing the optimizers. Necula proposed a method that checks the equality of the program semantics before and after the optimization by symbolic inference and evaluation [12]. It is fast and does not depend on each optimization. Rinard and Marinov proposed a method that infers the condition of the values of variables, so that the program semantics is preserved before and after the optimization. It also proves whether or not the program semantics is preserved after the execution of the optimization [13].

In contrast to these, our method belongs to the second approach, with a different strategy using CTL-FV. It is new in that it can be applied to existing optimizers. The result of optimization is correct if the system does not raise an alarm, in so far as we give the correct formulas.

8 Discussion

There are limitations to our method.

The check used in our method simply checks if the transformed points satisfy

the CTL-FV formulas written by the user. It does not validate if the semantics of the entire program is preserved in a strict sense.

We can check the preservation of semantics only if the respective formulas that the user manually writes are correct and complete. In this regard, no false alarms are reported.

The handling of exceptions is also manual.

However, we think the method of Lacey et al. [9], which proposed a method of proving the preservation of the program semantics, can be also utilized in our method. Proof of the preservation of the program semantics will be one of our future considerations.

Another limitation of our method is the type of optimization that can be handled. In general, our method can handle optimizations so that the properties of dataflow equations in the control flow graph after the optimization can be analyzed. In the COINS SSA optimization modules, rather complex optimizations are implemented, including operator strength reductions of induction variables and test replacements for induction variables using the SSA graph [6] and global value numbering based on question propagation and partial redundancy elimination [15]. These cannot be handled in the current framework. To check these optimizers by extending our proposed method will also be our future work.

9 Conclusion

We have proposed a method that checks if the execution of optimizers has preserved the program semantics, utilizing the model checking of CTL-FV. The proposed method describes, in terms of CTL-FV, a property that must hold to preserve the program semantics at each transformation point in the optimization of the program, and then uses model checking to confirm that the property holds after the execution of the optimizer.

Using this method, checks of various existing hand-written optimizers were performed using an aspect-oriented programming system. In addition, by implementing the proposed method and doing experiments, we found an unknown bug in an optimizer. The checking time was realistic, when compared to the compilation time.

Acknowledgments

This research was partially supported by the Japan Society for the Promotion of Science under the Grant-in-Aid for Scientific Research.

References

- [1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D.: “Compilers: Principles, Techniques, and Tools, 2nd ed.”, Addison-Wesley Longman Publishing Co., Inc., 2006.
- [2] Chiba, S., Nishizawa, M., and Kumahara, N.: GluonJ Home Page. <http://www.csg.is.titech.ac.jp/projects/gluonj/>.
- [3] Clarke, E. M., Emerson, E. A., and Sistla, A. P.: *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Trans. Prog. Lang. Syst., Vol. 8, No. 2 (1986), pp. 244–263.

- [4] Clarke, E. M., Grumberg, O., and Peled, 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 (2001), pp. 603–625.
- [7] 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 (1991), pp. 451–490.
- [8] Knoop, J., Rüthing, O., and Steffen, B.: *Optimal code motion: theory and practice*, ACM Trans. Prog. Lang. Syst., Vol. 16, No. 4 (1994), pp. 1117–1155.
- [9] Lacey, D., Jones, N. D., Wyk, E. V., and Frederiksen, C. C.: *Compiler optimization correctness by temporal logic*, Higher Order Symbol. Comput., Vol. 17, No. 3 (2004), pp. 173–206.
- [10] Lerner, S., Millstein, T., and Chambers, C.: *Automatically proving the correctness of compiler optimizations*, PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, ACM Press, 2003, pp. 220–231.
- [11] Lerner, S., Millstein, T., Rice, E., and Chambers, C.: *Automated soundness proofs for dataflow analyses and transformations via local rules*, POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 2005, pp. 364–377.
- [12] Necula, G. C.: *Translation validation for an optimizing compiler*, PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, ACM Press, 2000, pp. 83–94.
- [13] Rinard, M. and Marinov, D.: *Credible compilation with pointers*, Proceedings of the FLoC Workshop on Run-Time Result Verification, Trento, Italy, Jul. 1999.
- [14] Sassa, M. and Sahara, S.: *Validating Correctness of Compiler Optimizer Execution Using Temporal Logic*, Tech. Report C-247, Dept. Math. Comp. Sc., Tokyo Institute of Technology, 2007, <http://www.is.titech.ac.jp/research/research-report/C/index.html>.
- [15] 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>.
- [16] Wegman, M. N. and Zadeck, F. K.: *Constant propagation with conditional branches*, ACM Trans. Prog. Lang. Syst., Vol. 13, No. 2 (1991), pp. 181–210.