ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

Validating Correctness of Compiler Optimizer Execution Using Temporal Logic

Masataka Sassa
Soichiro Sahara

November 2007, C–247

Department of
Mathematical and
Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

Validating Correctness of Compiler Optimizer Execution Using Temporal Logic

Masataka Sassa and Soichiro Sahara
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology

November 19, 2007

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 testing whether or not the optimization transformations to the program have been performed correctly, by using temporal logic after the execution of the optimizer. We describe the properties that program points modified by the optimization have to satisfy to preserve the program semantics, in terms of temporal logic. Then the system 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.

1 Introduction

1.1 Background

Optimization in compilers is an important technique and is being actively investigated. However, optimizations are often complex, and bugs which 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.

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 [12] and Lerner et al.'s work [13, 14]. However, they cannot deal with complex optimizations. Verification studies include Rinard and Marinov's work [16] and Necula's work [15]. 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.

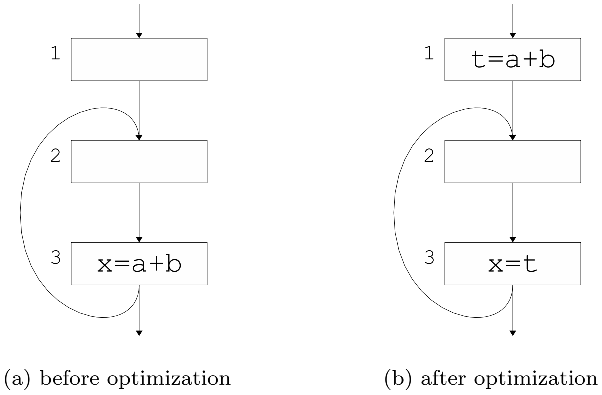
We use the temporal logic formula CTL-FV [12]. We first describe, in terms of CTL-FV, a property that each part of the program changed by optimization transformation must satisfy to preserve the program semantics. Then we check whether the property is satisfied, by model checking *after* execution of 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 [6]. As a result, we found an unknown bug in the optimizer for loop invariant code motion.

The details of the proposed method and consideration of its usefulness are presented in section 4 and section 6, respectively.



(a) before optimization

(b) after optimization

Figure 1: Example of 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.

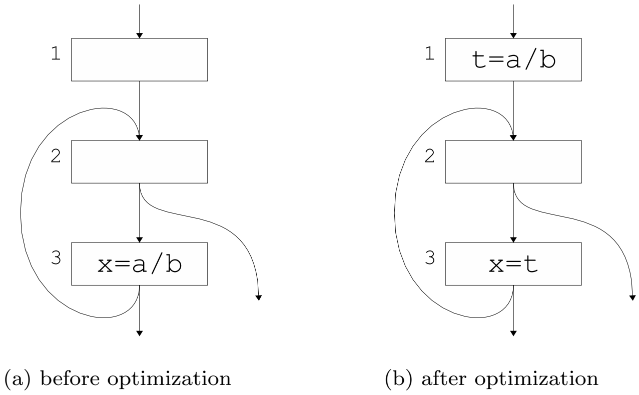
2.1 Correctness of optimization

What is required, in optimization, is at least to have the same program behavior before and after optimization. For example, if the return value of a function call differs before and after optimization, this will change the behavior of the program. 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. However, this is not a necessary condition for program semantics preservation, and we do not consider the issue in this paper. From now on, when we say “optimization is executed correctly”, this means that at least the optimized program preserves its semantics.



(a) before optimization (b) after optimization

Figure 2: Example of incorrect optimization of loop invariant code motion

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 $\mathtt{a} + \mathtt{b}$ is an expression whose value does not change inside the loop, we compute $\mathtt{a} + \mathtt{b}$ before the loop and assign its value to a temporary `t` as in Fig. 1(b). Use of the original $\mathtt{a} + \mathtt{b}$ 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 $\mathtt{a} + \mathtt{b}$, that is, inserting $\mathtt{a} + \mathtt{b}$ before the loop
- use of the original $\mathtt{a} + \mathtt{b}$ 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 $\mathtt{t} = \mathtt{a} + \mathtt{b}$ does not change the program semantics, the following condition must hold:

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

If the value of `t` is used at points other than replacement point, the original value of `t` at that point should be other than the value of $\texttt{a} + \texttt{b}$, so the program semantics will change.

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.

According to Aho et al.[1] and others, 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. [1]

Replacement of use of expression

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

- the values of $\texttt{a} + \texttt{b}$ 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.

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, 2].

Definition1 (Control flow model) We consider a control flow model $G = (N, E)$ 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 propositions is denoted by AP. For each node $n \in N$, we denote the mapping which gives the set of atomic formula $L(n)$ that holds at that node by the mapping $L\colon N \rightarrow 2^{AP}$. The triplet $M = (N, E, L)$ is called the *control flow model*. This is a Kripke structure [5], 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

[1] Strictly speaking this is not correct because this will change the number of calls of the signal handler which captures exceptions, but most optimizers rarely consider it.

$$
\begin{array}{lll}
L(n) = & & \\
& \{\ \mathit{node}(\mathtt{N}) \mid & n = \mathtt{N}\ ^{3}\} \\
\cup & \{\ \mathit{block}(\mathtt{B}) \mid & n \text{ is a statement in basic block } \mathtt{B}\ \} \\
\cup & \{\ \mathit{use}(\mathtt{X}) \mid & \text{variable } \mathtt{X} \text{ is used in } n\ \} \\
\cup & \{\ \mathit{def}(\mathtt{X}) \mid & \text{variable } \mathtt{X} \text{ is defined in } n\ \} \\
\cup & \{\ \mathit{comp}(\mathtt{E}) \mid & \text{expression } \mathtt{E} \text{ is computed in } n\ \} \\
\cup & \{\ \mathit{trans}(\mathtt{E}) \mid & \text{expression } \mathtt{E} \text{ is not changed in } n\text{, that is,} \\
& & \text{variables in } \mathtt{E} \text{ are not defined in } n\ \} \\
\cup & \{\ \mathit{mark}(\mathtt{M}) \mid & n \text{ has a mark } \mathtt{M}\ \}
\end{array}
$$

Table 1: Definition of $L(n)$

denoted by $n \rightarrow n'$. From now on, the term *model* refers to this control flow model.

Definition2 (Path on a control flow model) In a control flow model $M = (N, E, L)$, if an infinite sequence of states $n_0, n_1, n_2, ...$ 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 $n_0, n_1, ..., n_m$ 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*.

The definition of $L(n)$ in a control flow model $M = (N, E, L)$ is shown in Table 1. [2]

3.2 CTL-FV

The temporal logic used in our method is *CTL-FV* [12] proposed by Lacey et al. It is a logic based on CTL [4, 5], 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 propositions 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.

[2] The correct definition depends on the subject programming language. The definition for the COINS intermediate representation LIR can be found in [7].

[3] What is attached to node `1` is *node*(`1`).

$\phi ::= \mathit{true}$	$\phi ::= E\ \psi$	$\psi ::= X\ \phi$
$\phi ::= \mathit{false}$	$\phi ::= A\ \psi$	$\psi ::= \phi\ U\ \phi$
$\phi ::= \alpha$	$\phi ::= \overleftarrow{E}\ \psi$	$\psi ::= \phi\ W\ \phi$
$\phi ::= \neg\phi$	$\phi ::= \overleftarrow{A}\ \psi$	
$\phi ::= \phi \wedge \phi$		

Table 2: Syntax of CTL-FV

$\phi_1 \vee \phi_2$	$\equiv$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$
$\phi_1 \to \phi_2$	$\equiv$	$\neg\phi_1 \vee \phi_2$
$EF\ \phi$	$\equiv$	$E\,(\mathit{true}\ U\ \phi)$
$AF\ \phi$	$\equiv$	$A\,(\mathit{true}\ U\ \phi)$
$\overleftarrow{E}F\ \phi$	$\equiv$	$\overleftarrow{E}\,(\mathit{true}\ U\ \phi)$
$\overleftarrow{A}F\ \phi$	$\equiv$	$\overleftarrow{A}\,(\mathit{true}\ U\ \phi)$

Table 3: Syntax sugar of CTL-FV

- It is possible to perform model checking efficiently.

Syntax of CTL-FV

The syntax of CTL-FV is shown in Table 2. ϕ is a nonterminal deriving expression concerning states (*state expressions*), ψ is a nonterminal deriving expression concerning paths (*path expressions*), and α is a proposition 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 3.

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 4 [4].

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

[4] Strictly speaking, the semantics is defined on formulas by binding all free variables in the formulas by the symbols of the program.

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 $n \models \phi_2$
$n \models E\psi$	iff	$\exists p = n \rightarrow n_1 ..., p \models \psi$
$n \models A\psi$	iff	$\forall p = n \rightarrow n_1 ..., p \models \psi$
$n \models \overleftarrow{E}\psi$	iff	$\exists p = n \rightarrow^{\circ} n_1 ..., p \models \psi$
$n \models \overleftarrow{A}\psi$	iff	$\forall p = n \rightarrow^{\circ} n_1 ..., p \models \psi$
path formula ($p = n_0 \rightarrow' n_1 ...$, $\rightarrow'$ is $\rightarrow$ or $\rightarrow^{\circ}$)		
$p \models X\phi$	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+i} exists])

Table 4: Semantic definition of CTL-FV

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.

The check used in our method simply checks if the transformed points satisfy the CTL-FV formulas. It does not validate if the semantics of the whole program is preserved in a strict sense. However, we think the method of Lacey et al. [12],

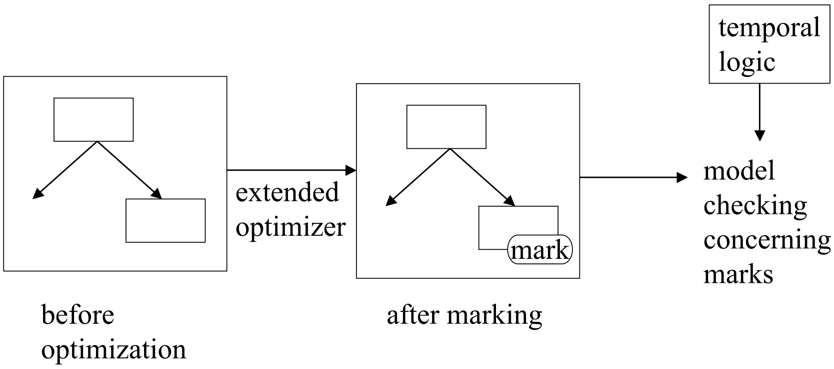


Figure 3: Outline of the proposed method

which proposed a method of proving preservation of the program semantics, can be also utilized in our method. Proof of preservation of the program semantics will be one of our future considerations.

4.2 Steps to realization of the proposed method

The proposed method comprises the following five steps:

■ Preparation:

1. For each optimization, describe the condition for the correctness of the optimization, as a *specification* to be checked.

2. Extend the existing optimizer so that marks can be added to points transformed during optimization.

■ Before optimization:

3. Make the model of the program.

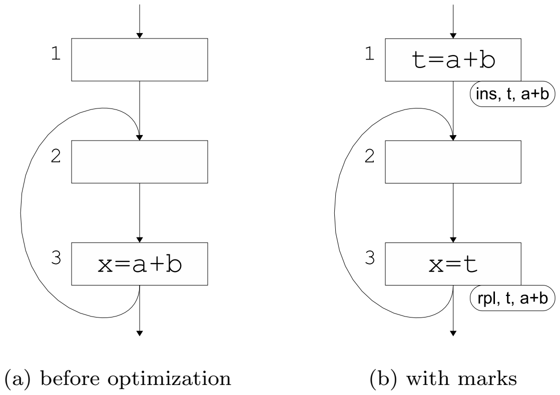
■ During the execution of the optimization:

4. Mark the points of the program transformed by the optimizer.

■ After the optimization:

5. 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 $(\mathtt{ins}, \mathtt{t}, \mathtt{a} + \mathtt{b})$, and the replacement point is marked by $(\mathtt{rpl}, \mathtt{t}, \mathtt{a} + \mathtt{b})$, showing the type of transformation at each point.
Step1 : Description of the checking specification



(a) before optimization

(b) with marks

Figure 4: Example of loop invariant code motion (with marks)

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 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 $\langle m, \phi \rangle$.

In the following, we consider the description of the checking specification for the loop invariant code motion of Fig. 4.

The point where the hoisted expression is inserted

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

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

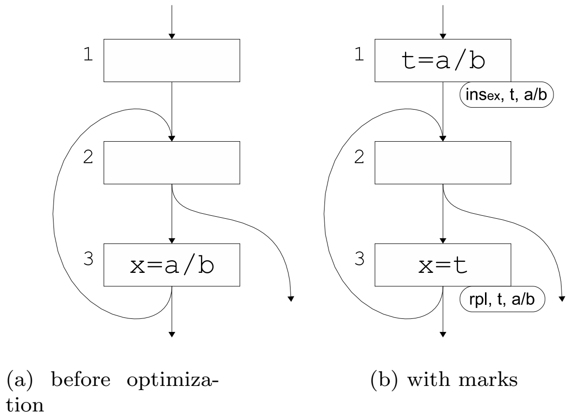
In other words,

- 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 $\mathtt{a} + \mathtt{b}$ by the free variable e, and consider that the replacement point is marked by $(\mathtt{rpl}, t, e)$, the CTL-FV formula expressing this is:

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

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



(a) before optimization

(b) with marks

Figure 5: An example of erroneous loop invariant code motion (with marks)

In cases where an expression which may cause an exception, such as `a/b`, is hoisted and inserted, the following condition, presented in section 2.2, 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 $(\texttt{ins}_{\texttt{ex}}, t, e)$, 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 is[5]:

$$A\,(\mathit{true}\ W\ \mathit{mark}(\texttt{rpl}, t, e)) \tag{2}$$

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

The point where use of an expression is replaced

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

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

In other words,

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

[5] The reason why we use a W operator instead of a U operator is to avoid considering paths that do not reach the exit point, such as an infinite loop.

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 `t` by a free variable t and an expression $\texttt{a} + \texttt{b}$ by a free variable e, the replacement point is the node marked by either the mark $(\texttt{ins}, t, e)$ or the mark $(\texttt{ins}_{\texttt{ex}}, t, e)$. Therefore, the CTL-FV formula is[5]:

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

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

Step2 : Extension of the existing optimizer

In our method, in order to realize the marking to the *transformed part of the program*[6] during the execution of the optimizer, we need to insert code for adding marks into those sections of the *source code 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 [3].

Step3 : Modeling of the program

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.

There might be an implementation of the optimizer such that, when deleting a statement, the node is replaced by a "skip" or "nop" statement, which means "do nothing", instead of actually deleting the node. However, most optimizers generally delete the node itself in the control flow graph in such cases, so we will have to handle this issue.

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

[6] The program may be in an intermediate form such as a control flow graph.

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:[7]

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

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 $(\mathtt{ins}, \mathtt{t}, \mathtt{a} + \mathtt{b})$ is formula (1). If we consider the correspondence between the actual mark and formula (1), $\mathtt{t}$ corresponds to t and $\mathtt{a} + \mathtt{b}$ corresponds to e. Therefore, we bind formula (1) using them. This gives formula (5):

$$\neg E\,(\neg \mathit{def}(\mathtt{t})\ U\ (\mathit{use}(\mathtt{t}) \wedge \neg \mathit{mark}(\mathtt{rpl}, \mathtt{t}, \mathtt{a} + \mathtt{b}))) \tag{5}$$

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

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

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

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

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

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

According to Fang and Sassa [10], when we want to generate optimizers using CTL-FV formulas, the number of combinations of binding of free variables greatly influences the time required for model checking. However, in our method, the binding is made using actual marks. Therefore, it is possible to reduce the number of combinations to only a few.

There are several ways of performing model checking. Our implementation used the algorithm based on classical CTL model checking given in [4].

[7] The meaning is the same if we omit n and write $\models \mathit{mark}(m) \rightarrow \phi$.

5 Application to Actual Optimizers

In the backend of the COINS compiler [6], many optimizers that handle its Low-level Intermediate Representation (LIR) [7] 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][11]

Hereafter, we will describe details of the application to loop invariant code motion and constant propagation with conditional branches in the SSA form.

5.1 SSA form

An SSA form is a program representation in which the definition of each variable appears only once in the program text [2, 9]. The SSA form is said to be favorable for program optimization.

The SSA form has the following advantageous properties:

- Only one definition reaches a use of a variable.
- When different definitions of a variable v merge at a node n in the control flow graph, a ϕ *function* is inserted at the beginning of n, which can distinguish the values of v reaching that point.

5.2 Loop invariant code motion

In COINS, loop invariant code motion in the SSA form is implemented. This optimization is effectively the same as the loop invariant code motion in the normal (non-SSA) form explained in section 2.2 and section 4.2. It comprises two transformations:

- Hoist the loop invariant expression and insert it before the loop.
- Replace the use of the loop invariant expression by a temporary variable.

[8] Not included in the standard COINS distribution.

However, the properties that these transformed points must satisfy to preserve the program semantics are slightly different from those in section 4.2. They are properties utilizing the characteristics of the SSA form.

Point where hoisted expression is inserted

In the SSA form, redefinition of variables is not allowed. Therefore, when $t = e$ is inserted, the condition of correctness is now that t is not defined in other nodes, instead of formula (1). If we denote the node where $t = e$ is inserted by n, the CTL-FV formula which must hold at the insertion-point node n is:

$$\neg EF\ (\neg node(n) \wedge def(t))\ \wedge \neg \overleftarrow{E}F\ (\neg node(n) \wedge def(t)) \tag{9}$$

In other words, this is a formula that specifies that there are no nodes defining t other than n. In addition, if e is an expression which may cause an exception, such as `a/b`, formula (2) is necessary as a condition of correctness, similarly to the normal-form case.

Point where use of expression is replaced

The condition for the point where expression e is replaced by temporary variable t is sufficient if we use formula (3), as in the case for the normal form. However, for SSA form, it is guaranteed by formula (9) that t is not defined at any position other than the insertion point. Therefore, $\neg def(t)$ in formula (3) can be omitted. Then, the CTL-FV formula to be satisfied at the node of the replacement point can be written as:

$$\overleftarrow{A}\ (trans(e)\ W\ (mark(\mathtt{ins}, t, e) \vee mark(\mathtt{ins_{ex}}, \mathtt{t}, \mathtt{e}))) \tag{10}$$

As described above, the proposed method can similarly handle both optimization for the normal form and optimization for the SSA form, by paying attention to the difference between their properties.

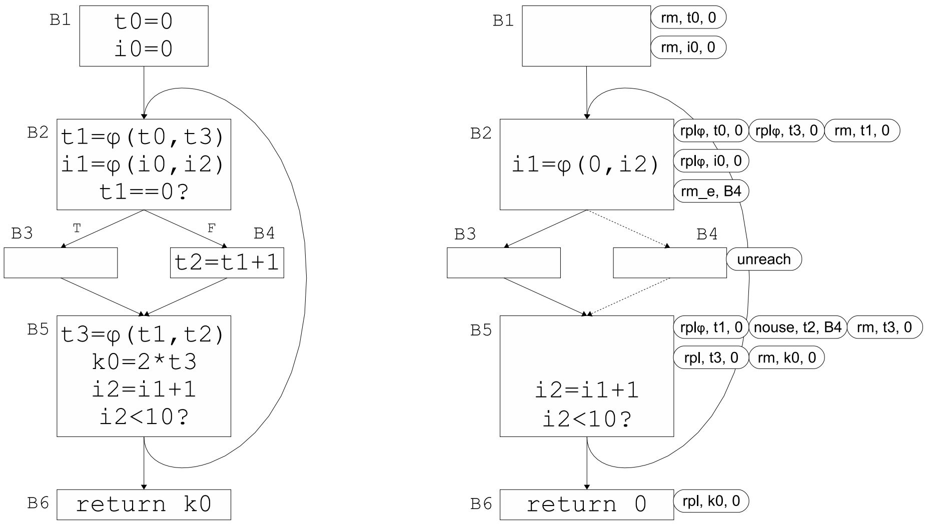
5.3 Constant propagation with conditional branches

COINS implements a powerful constant propagation algorithm in SSA form called “constant propagation with conditional branches” [20].

In this section, a node of the control flow graph may contain several statements instead of one statement. Fig. 6 is an example in which constant propagation with conditional branches is applied. If we traverse the graph of Fig. 6(a) in sequence from the entry point, we know that `t1` always has the value `0` and block `B4` is unreachable. This optimization is not based on analysis using dataflow equations. Rather, it is an optimization based on the framework of the *abstract interpretation*, in which values are checked by traversing the control flow graph.

After optimization, the program becomes as shown in Fig. 6(b). The use of variables that are known to always have constant values, such as `t0`, `t1`, and `k0`, are replaced by constants, and assignments to these variables are eliminated. Furthermore, statements in unreachable blocks such as `B4` and branches to those blocks are eliminated.

We consider that this optimization of constant propagation with conditional branches cannot be performed by the method of Lacey et al. However, with our method, we can check whether this optimization *was correctly executed.* The reason will be explained in section 5.3.2.



(a) before optimization

(b) after optimization (with marks)

Figure 6: Example of constant propagation with conditional branches

5.3.1 Correctness of constant propagation with conditional branches

The set of transformed points of constant propagation with conditional branches can be classified into the following five types:

1. The point where an assignment is deleted because the value of variable x becomes a constant c by assignment to x. We mark it by $(\texttt{rm}, x, c)$.

2. The point where the use of variable x is replaced by constant c. We mark it by $(\texttt{rpl}, x, c)$.

3. The point where a branch to block b is deleted. We mark it by $(\texttt{rm\_e}, b)$.

4. The point where statements in the node of an unreachable block are deleted. We mark it by $(\texttt{unreach})$.

5. The point where a parameter of a ϕ function corresponding to an unreachable block p is deleted. We mark it by $(\texttt{nouse}, x, p)$.

In the following, we briefly present the correctness of these transformations and give their specifications that we described.

Point where assignment of constant is deleted

If the right-hand side e of assignment $x = e$ can be evaluated to constant c, the value of x is c, and all later use of x can be replaced by c. Therefore, this assignment can be deleted.

The method of evaluation differs depending on whether or not the right-hand side e is a ϕ function:

- In the case when e is a ϕ function, e is evaluated to c, if each variable used in e has the constant value c or is a variable evaluated when the control flow comes from unreachable nodes.

- In the case when e is not a ϕ function, if the variables used in e are all constants[9], and the result of evaluation of the expression is c, e is evaluated to c.

Note that if x is a variable whose value is a constant c, the node where this statement resides is marked by $(\texttt{rpl}, x, c)$. Note also that if variable x in a ϕ function is a variable coming from an unreachable block p, the node where this statement resides has the mark $(\texttt{nouse}, x, p)$.

This method of evaluation is defined in the function *eval* as follows:

$$\begin{array}{rcl}
eval(c) & = & c \\
eval(x) & = & \begin{cases} \top & \text{if marked } (\texttt{nouse}, x) \\ c & \text{if marked } (\texttt{rpl}, x, c) \\ \bot & \text{otherwise} \end{cases} \\
eval(x\, op\, y) & = & \begin{cases} c & \text{if } eval(x) = c_1 \\ & \wedge\ eval(y) = c_2 \\ & \wedge\ c_1\, op\, c_2 = c \\ \bot & \text{otherwise} \end{cases} \\
eval(\phi(x_1, ..)) & = & \begin{cases} c & \text{if } \forall x_i, eval(x_i) = c \\ & \vee\ eval(x_i) = \top \\ \bot & \text{otherwise} \end{cases}
\end{array}$$

Here, $\top$ is a symbol indications that the variable may be some as yet undetermined constant (undetermined), and $\bot$ is a symbol indicating that a constant value cannot be guaranteed (indefinite) [20]. In addition, c, c_1, and c_2 refer to constants, x and y refer to variables, and *op* refers to an arithmetic operation.

If $eval(e) = c$, the statement $x = e$ can be deleted, and the node is marked by $(\texttt{rm}, x, c)$. The formula checked at this node is:

$$eval(e) = c \tag{11}$$

Strictly speaking, this formula is not a CTL-FV, as defined in section 3.2. However, it can be checked at the time of marking by investigating the statements of the node and the marks attached at that time, and needs no validation by model checking. Therefore, we perform this check concerning the marks by computing $eval(e)$ at the time of marking.

Point where use of variable is replaced by constant

Use of variable x at a node, where the value of x is the constant c, can be replaced by c. The condition that the value of x is c at a node n is that there is a node m equivalent to the statement $x = c$ in a predecessor path, and m dominates n. At a node m where a statement equivalent to $x = c$ resides, the statement is deleted, and the mark $(\texttt{rm}, x, c)$ is attached.

If we also consider deleting an unreached block, it suffices that, in all predecessor paths of n, a node m that dominates node n exists or an unreachable block exists. Therefore, the formula to be checked at node n is:

$$\overleftarrow{A}\ (true\ W\ (mark(\texttt{rm}, x, c) \vee mark(\texttt{unreach}))) \tag{12}$$

A little more care is needed when replacing parameter x of a ϕ function by a constant c. If x is a parameter evaluated when control flows from a predecessor

[9] Cases such as x in $0 \times x$ can be handled specially.

block p, the last statement of p must be dominated by a node m that contains a statement equivalent to $x = c$, but the last statements of predecessor blocks other than p may not be dominated by m. For example, in Fig. 6, `t3` is replaced by `0` at $\texttt{t1} = \phi(\texttt{t0}, \texttt{t3})$ of block `B2`. `t3` must be defined to have the value `0` when control comes from `B5`, but it may undefined if control comes from `B1`. From the above, if we denote by b the block where node n resides, and consider that another ϕ function node may exist in b before n, replacement of x by c is correct if the following condition holds:

- In all reverse paths from n, a node of block b (this is the node of the ϕ function) may appear several times, and after these appearances, a node of another block (this is the predecessor block of b) appears. If the latter block is p, the formula (12) must hold at the last node of that block.

If we put a special mark $(\texttt{rpl}_{\phi}, x, c)$ at the replacement point of the parameter x of a ϕ function, the formula to be checked at node n with mark $(\texttt{rpl}_{\phi}, x, c)$ attached is:

$$\overleftarrow{A}\ (\mathit{block}(b)\ U\ (\mathit{block}(p) \rightarrow \phi)) \tag{13}$$

Here, ϕ is formula (12).

Point where branch is deleted

If the conditional expression e of a conditional branch can be evaluated to either *true* or *false*, the branch can be decided at compile time, and an unnecessary branch can be deleted. Similarly to the handling of points where constant assignment statements are deleted, this check can also be made at the time of marking. This is similar to the computation of the *eval* function, and model checking is not necessary.

Point where statements of unreachable block are deleted

If there are no edges to block b, b is an unreachable block. We mark nodes in b by (`unreach`). If all branches to b are deleted, b becomes an unreachable block. A node in an unreachable block satisfies one of the following conditions:

- If the node is at the beginning of a block, the previous node should have been a node that is a statement where a branch was deleted (it is marked by $(\texttt{rm\_e}, b)$).

- If the node is not at the beginning of a block, the previous node is a node whose statement was deleted. The node whose statement was deleted should have been either an unreachable node (marked by (`unreach`)) or a node deleted because it was a constant assignment statement (marked by $(\texttt{rm}, x, c)$).

From the above, the formula to be checked at a node marked by (`unreach`) is:

$$\overleftarrow{A}X\ (\mathit{mark}(\texttt{rm\_e}, b) \vee \mathit{mark}(\texttt{unreach}) \vee \mathit{mark}(\texttt{rm}, x, c)) \tag{14}$$

Point where a parameter from an unreachable node in a ϕ function is deleted

All parameters of a ϕ function are related to some predecessor block. If a parameter x of a ϕ function is related to a predecessor block p, this parameter can be deleted if either p is an unreachable block or if the last node of p is the node where a branch to block b, in which this ϕ function resides, is deleted. If we denote the block where this ϕ function resides by b, the formula to be checked at the node marked by $(\texttt{nouse}, x)$ is the following, using similar considerations to the case of formula (13):

$$\overleftarrow{A}\,(block(b) \quad U \quad (block(p) \quad \rightarrow \quad (mark(\texttt{unreach}) \;\vee\; mark(\texttt{rm\_e}, b))) \tag{15}$$

5.3.2 The reason why checking is possible by our method

Constant propagation with conditional branches is not an optimization that can be performed by solving the dataflow equation on the control flow graph. Rather, it is an optimization based on an abstract interpretation of the program. In a method whereby a dataflow equation on the flow graph is solved, there is the premise that all edges may be executed. Therefore, we cannot deduce that `t1`, for example, is a constant in any execution, and we cannot deduce that `B4` is unreachable.

Dataflow equations are said to have a computational power equal to the μ computation [18]. The method of Lacey et al. uses model checking by CTL-FV, whose computational power is less than μ computation, for the program analysis. Therefore, it cannot handle this optimization.

By contrast, our method only checks whether the *result* of transformation by the optimization, as shown in Fig. 6(b), is truly correct. This can be performed by the model checking of CTL-FV. In other words, we are given the result of the analysis that node `B4` is unreachable and variable `t1` always has the value `0`. Therefore, the formulas to check whether they are correct can be described as in section 5.3.1.

6 Experiments and Discussion

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 [19]. 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:

1. During checking, a counterexample was detected in the model checking of the specification corresponding to mark $(\texttt{ins}_{\texttt{ex}}, t, e)$. 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 5: Number of extension points for each optimizer

2. 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 (the part of the program or the control flow graph changed by the optimizer) 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.[10]

Table 5 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 5, 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 distinctive[11]. Therefore, it is easy for authors who have a general knowledge of optimizers to find them.

However, for a checking user who does not have much knowledge of optimizers, it will not be easy to find the extension points. It is an issue for the future to devise a technique that finds the extension points automatically, or enables them to be found easily.

[10] In our research, we did not complete the implementation of dead code elimination, but the number of extension points and rewriting points would not be increased.

[11] For example, method calls that operate on the control flow graph.

CPU	Intel Pentium 4 CPU 2.80GHz
OS	Linux 2.6.17-13msmp
JavaVM	1.5.0_08
Heap Size	256Mbyte
Stack Size	2048Kbyte
COINS	1.4.1
Benchmark	SPEC CPU 2000 version 1.2

Table 6: Environment for the experiments

6.3 Cost of describing the checking specification

In our method, we need to describe the specification to be checked, corresponding to each type of mark for each optimizer. Here, we discuss how much effort is needed to describe the checking specification.

Amount of description

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 5. 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

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.

Therefore, not all users can easily write a description for the checking specification, However, considering the short length of the description, this will not be too difficult if the user knows the optimizer relatively well and is accustomed to temporal logic. In our case, writing the checking specification of the six optimizers in Table 5 required about three weeks, including the time to understand the precise behavior of the optimizers from a reading of the source code. The reason why one of the authors should read not only the specification but also the source code of the optimizers is that there were some differences between the specification and the actual implementation.

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, using our method. The environment for the experiments is shown in Table 6.

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.

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

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. (The SSA optimizations performed when option O2 is specified comprises common subexpression elimination, constant propagation with conditional branches, global value numbering by question propagation and partial redundancy elimination, operator strength reduction and test replacement for induction variables, copy propagation, loop invariant code motion, and dead code elimination.) [12]

The measurements are shown in Table 7. 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.[13] 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, as follows. In our method, we perform model checking after binding the free variables in the CTL-FV formula of the checking specification, as explained in section 4.2. This means that, from a single checking specification, checking formulas for as many as the number of attached marks are generated. We think

[12] Lazy code motion is not included because it was only at the prototype stage.

[13] 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.

that the formulas generated from a single specification could be checked together instead of checking them individually, similarly to the bit-vector method in dataflow equations [2]. Just as the computation of dataflow equations can be made very rapid by the bit-vector method, we might expect that the checking in our method could also be made much more rapid.

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

7 Related Work

Much research concerning the correctness of optimization exists. In this section, we compare our method with other research that is closely related to our method.

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 [12]. They also showed that, for a few optimizers, it is possible to prove simply that the optimizers realized by their method preserve the program semantics. The proven optimizers are guaranteed to always execute correctly, and are bug free. However, their method can realize only the relatively simple optimizers that can be described by dataflow equations alone. For example, the constant propagation with conditional branches presented in section 5.3 can be handled by our method, but cannot be handled by the method of Lacey et al.

Lerner et al. proposed a system that describes the optimizer using an original domain-specific language based on temporal logic, and executes the optimizer. [13, 14]. This is similar to the system of Lacey et al., but it has the feature of being able to perform the proof of the correctness of the optimizer almost automatically, by using a theorem prover. However, the system of Lerner et al. cannot handle complex optimizations, similarly to Lacey et al.'s method.

Necula proposed a method that checks the equality of the program semantics before and after the optimization by symbolic inference and evaluation [15]. This checking method does not depend on the optimization, in principle. Therefore, the cost of its realization would seem to be lower than our method, which makes a checking specification for each optimization. In addition, the checking time efficiency seems to be high. However, in their method, there are cases where inference or evaluation fails, and there is the defect that strict preservation of the program semantics is not guaranteed. Our method also does not guarantee the strict preservation of the program semantics, but we think it may be possible to give a strict proof. Moreover, after a bug is found, identification of its cause in Necula's method seems more difficult than in our method.

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 [16]. This method seems to guarantee the preservation of strict program semantics, but it is unclear how practical it might be, because no algorithms have been written.

8 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 optimizers were performed. 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.

Future work will mainly involve the following three items:

Strict proof

The specification of transformation points of optimization described by our method does not guarantee the correctness of optimization by itself. Many of them are intuitively trivial, or are already proved in other references. However, strictly speaking, we must correctly define the formal semantics of the subject program, and then give a proof that "it is correct if it satisfies the specification." Lacey et al. did this by hand, and Lerner et al. used a theorem prover. We think our method can perform this strict proof similarly to them.

Application to more complex optimizations

In the COINS SSA optimization modules, rather complex optimizations are implemented, including operator strength reduction of induction variables and test replacement for induction variables [8], and global value numbering based on question propagation and partial redundancy elimination [17]. We suspect that there is a bug in at least one of these optimizers. We want to check these optimizers using the proposed method.

Making the check faster

It was not an aim of this paper to achieve rapid model checking, so little consideration was given to making the detailed algorithms faster. However, we expect to make the checking time much shorter by using the method presented in section 6.4.

Automatically finding the extension points

As suggested in section 6.2, finding the extension points in the optimizer will be difficult when the checking user does not have sufficient knowledge of optimizers. A method for automatically finding the extension points or finding them more easily is one of the problems for the future.

Acknowledgments

This research was partially supported by the Japan Society for the Promotion of Science under the Grant-in-Aid for Scientific Research, and by the Okawa Foundation for Information and Telecommunications.

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] Appel, A. W. and Palsberg, J.: "Modern Compiler Implementation in Java, 2nd ed." Cambridge University Press, 2002.

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

[4] 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.

[5] Clarke, E. M., Grumberg, O., and Peled, D. A.: "Model Checking", The MIT Press, 1999.

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

[7] COINS Project: COINS project LIR specification (in Japanese), 2002. `http://www.coins-project.org/spec/lir.pdf`.

[8] 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.

[9] 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.

[10] Fang, L. and Sassa, M.: *Generating Java compiler optimizers using bidirectional CTL*, Electronic Notes in Theoretical Computer Science, Vol. 190/4 (2007), pp. 49–63.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Rinard, M. and Marinov, D.: *Credible compilation with pointers*, Proceedings of the FLoC Workshop on Run-Time Result Verification, Trento, Italy, Jul. 1999.

[17] 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`.

[18] Schmidt, D. A.: *Data flow analysis is model checking of abstract interpretations*, POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 1998, pp. 38–48.

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

[20] 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.