ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

Experience in Testing Compiler Optimizers Using Comparison Checking

Masataka Sassa
Daijiro Sudo

February 2006, C–221

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

Experience in Testing Compiler Optimizers Using Comparison Checking

Masataka Sassa and Daijiro Sudo*

February 4, 2006

Abstract

This paper describes our experience of testing and debugging an optimizer using comparison checking. Although this study is based on Jaramillo et al.'s work, the experience will help those who test optimizers using this technique.

In our implementation, important values during the execution of programs are output as a file trace before and after each optimization. Then a comparison phase checks these results. When the comparison checker finds an error in the optimizer the system shows a C language style program that is back translated from the intermediate code. Therefore, the optimizer writer can easily find the erroneous section of the optimizer. We have implemented the system on the compiler infrastructure COINS, and have verified the optimizers based on static single assignment (SSA) form that our group has been developing. The practical treatment of several related issues is also described.

By applying this technique, we found four bugs, including two unknown bugs, in the optimizer in an existing compiler project.

Keywords: compiler optimization, testing, debugging, comparison checking.

1 Introduction

Considerable effort is required to confirm correctness of optimizers in compilers. We describe our experience in testing and debugging optimizers using comparison checking. Although this study is based on Jaramillo et al.'s work [2], the experience will help those who test optimizers using this technique.

In our implementation, important values during execution, such as the left-hand side of assignment statements or both sides of the relational operator in conditional expressions, are output to files as the trace. This is done before and after each optimization. Then a comparison phase checks these traces. When the comparison checker finds an error in the optimizer it is generally difficult to find the cause of the failure. Our system produces a C language style program that is back translated from the intermediate code. Therefore, the optimizer writer can easily find the erroneous section of the optimizer.

We have implemented the system on the compiler infrastructure COINS [3] and have verified the optimizers based on the static single assignment (SSA) form [1] that our group has been developing. Although using the SSA form simplifies finding correspondence between variables before and after optimizations, the method itself does not depend on the SSA form.

*M. Sassa and D. Sudo are with Tokyo Institute of Technology. Email: sassa@is.titech.ac.jp.

For optimization and transformation, we treat machine-independent optimization: copy propagation, constant propagation, common subexpression elimination, common subexpression elimination based on question propagation, partial redundancy elimination, loop invariant code motion, strength reduction and test replacement of induction variables, dead code elimination, removal of unnecessary ϕ-functions, and removal of empty blocks.

We found four bugs in the optimizers, including two previously unknown bugs, in an existing compiler project. It proved to be effective in use.

We describe the method of the comparison, which can deal with the situation where instructions may be deleted, or moved, by the optimizer. The practical treatment of several related issues, such as the treatment of false alarms, is also described.

2 Testing Optimizers Using Comparison Checking

In this section, we outline the method of Jaramillo et al. for testing optimizers using comparison checking [2].

The method runs the programs before and after optimization alternately similar to coroutines, using the same input, and checks the behavior of both programs, mainly by comparing the values of corresponding variables. If the value of a variable is different in some part of both executions, it displays the variable and the earliest program point where the values are different.

The comparison checking is done in three phases: (1) determine which values computed by both programs need to be compared, (2) determine where the comparisons are to be performed in the program execution, and (3) perform the comparisons. To achieve these tasks, three sources of information, mappings, annotations, value pools, are used.

Determine the values to be compared To compare values computed in the program before and after optimization, the correspondence of statements whose value must be the same in both programs must be identified, which is done using *mappings*.

Determining where the comparison is to be made Identification of program points where the comparison checks should be performed is done by *annotation*.

Comparison checking In Jaramillo et al.'s method, the execution of the unoptimized program controls checking and the execution of the optimized program. Execution proceeds until a breakpoint is reached; at each breakpoint the control is transferred to the other program and/or the checker compares the value of corresponding variables. The programs before and after optimizations are therefore executed alternately, similar to coroutines. Because there may be statements moved by some optimization *value pool* is used to temporarily save these values.

3 Implementation and Techniques of Comparison Checking to Test Optimizers

In this section, we describe our implementation and techniques of comparison checking to test optimizers. Our method differs somewhat from Jaramillo et al.'s and the following features are distinctive:

1. alternate execution is not used and no breakpoints are used

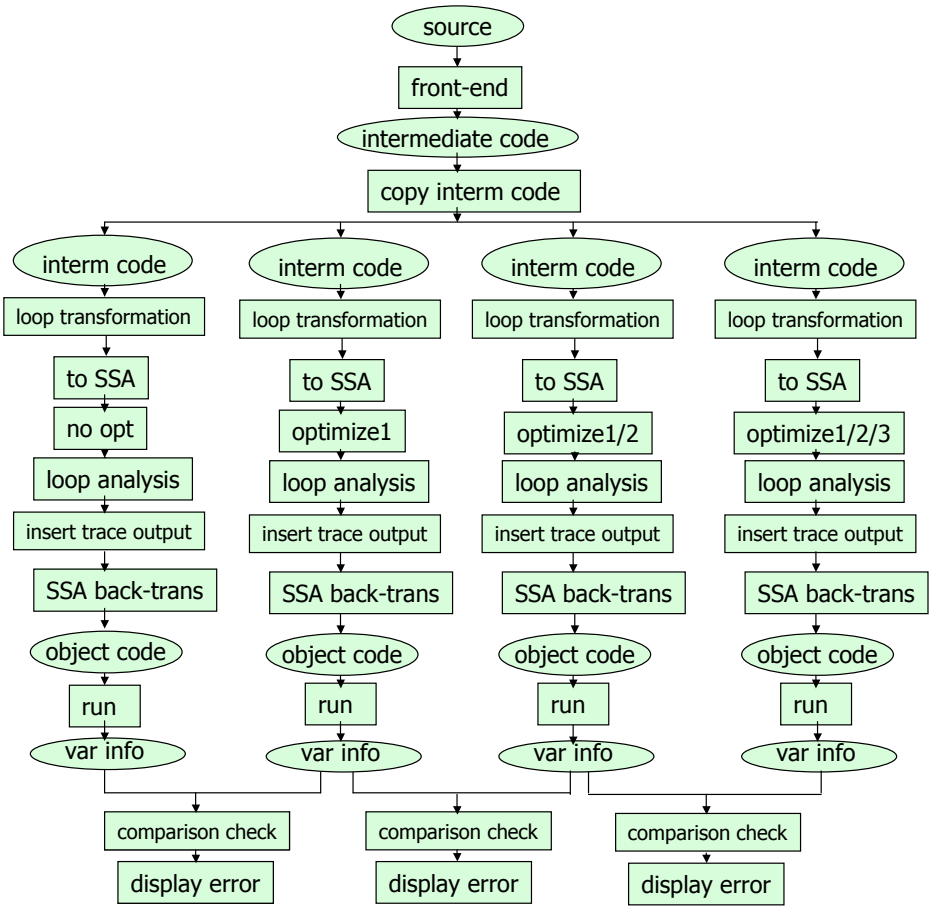

Figure 1: Outline of the method (opt1, opt2, opt3 are SSA optimizations)

2. trace files are generated
3. the intermediate code is displayed in the case of failure
4. false alarms by pointers are managed
5. SSA form is used
6. other implementation issues, such as handling of natural loops, checking conditional expressions, and handling of each type of optimization are described

3.1 Outline

The outline of our implementation is shown in Figure 1. In our implementation, the compiler is extended as follows.

First, the intermediate code is copied to make (*number of optimization passes* + 1) copies. Next, for each copy of the intermediate code, loop transformation of natural loops is applied. Then, the intermediate code is transformed into SSA form, SSA form optimizations (called SSA optimizations below) are applied. Loop analysis and the insertion of trace output are performed. Then SSA back-translation is done to produce normal form intermediate code. The object codes generated by the extended compiler are executed with input, and the trace,

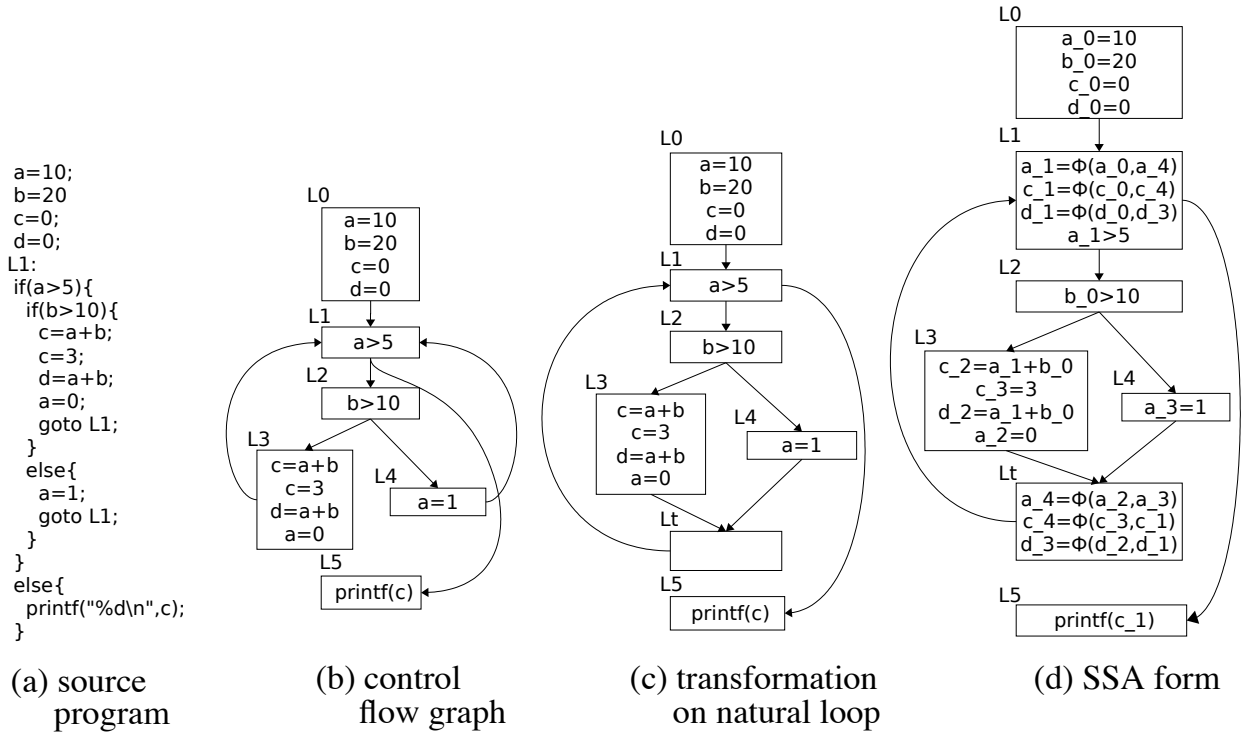

Figure 2: Example program and control flow graph

consisting of variables and their values (called *variable information*), is written to file. Finally, comparison checking is achieved based on the trace "variable information", and the error points displayed if comparison errors occur.

A "variable information" is a trace of the values of variables etc. at execution time, and contains the following: (i) basic block number (ii) instruction number in the basic block (iii) variable name etc. (the left-hand side of the assignment statement, both sides of the comparison of a conditional expression, parameters of function call) (iv) value (of variable name etc.). The "variable information" also contains *loop information*, which is tags representing the entry block and exit block of each loop. Variable names etc. for conditional expressions and function calls are described in subsection 3.2.6.

If comparison errors occur, the system displays an error message including the basic block number, the instruction number in the basic block, variable names etc. and their values in the program before and after optimization.

The optimized intermediate code is quite different from the source program. To bridge the gap between the optimized intermediate code and the source program, our system back-translates the intermediate code into a C-style program and displays it. By examining this code, the optimizer writer can easily determine the nature of any erroneous transformation caused by the optimizer.

In the following explanation, we use the example program shown in Figure 2(a). Its control flow graph (CFG) is shown in Figure 2(b).

3.2 Algorithm

3.2.1 Duplication of intermediate code source program

We make *(number of optimization passes + 1)* copies of the intermediate code. Optimization passes are applied sequentially for each copy of the intermediate code. Thus, we can check which optimization pass is correct or erroneous after comparison checking.

3.2.2 Loop transformation of natural loops

We assume that all loops are natural loops to facilitate comparison checking. Even a natural loop may contain loops with a common header that are not nested in each other. In that case, we merge these loops by adding an empty basic block.

An example is shown in Figure 2(b). In Figure 2(b), there are two loops LOOP1(L1,L3) and LOOP2(L1,L4). However, these two loops are not nested within each other and have a common header. Comparison checking is difficult for these loops, therefore, such loops are merged into one loop. The example CFG of Figure 2(b) is transformed into Figure 2(c) by adding a new empty basic block Lt. Therefore, the original two loops LOOP1(L1,L3) and LOOP2(L1,L4) are merged into one loop LOOP(L1,Lt).

3.2.3 SSA translation

Applying SSA translation [1] to the program of Figure 2(c) produces Figure 2(d).

3.2.4 SSA form optimizations

Optimizations (including transformations in a broad sense) that can be handled in our method are: copy propagation, constant propagation, common subexpression elimination, common subexpression elimination based on question propagation, partial redundancy elimination, loop invariant code motion, strength reduction and test replacement of induction variables, dead code elimination, removal of redundant ϕ-functions, and removal of empty basic blocks.

Figure 3(a) (b) and (c) are the results of optimizing Figure 2(d). Figure 3(a) is the result of applying loop invariant code motion to Figure 2(d). Basic blocks L0, L3 and L4 are changed. Figure 3(b) is the result of applying common subexpression elimination. Basic block L3 is changed. Figure 3(c) is the result of further applying dead code elimination. Basic blocks L0, L1, L3 and Lt are changed.

3.2.5 Loop analysis

The current implementation treats SSA optimizers. In the SSA form, although the definitions of variables are textually unique, the same SSA variable in a loop is usually assigned more than once at runtime and therefore appears several times in the output trace. In optimizations involving code motion such as loop invariant code motion, the system should know which loop is the subject of optimization. Therefore, loop analysis is performed using the utility functions (Java methods) for loop analysis provided in COINS and leaves the result as tags of the loop.

3.2.6 Insertion of statements for outputting trace

The system inserts trace output statements in the intermediate code for outputting the values of variables etc., which are assigned at runtime, to the trace file.

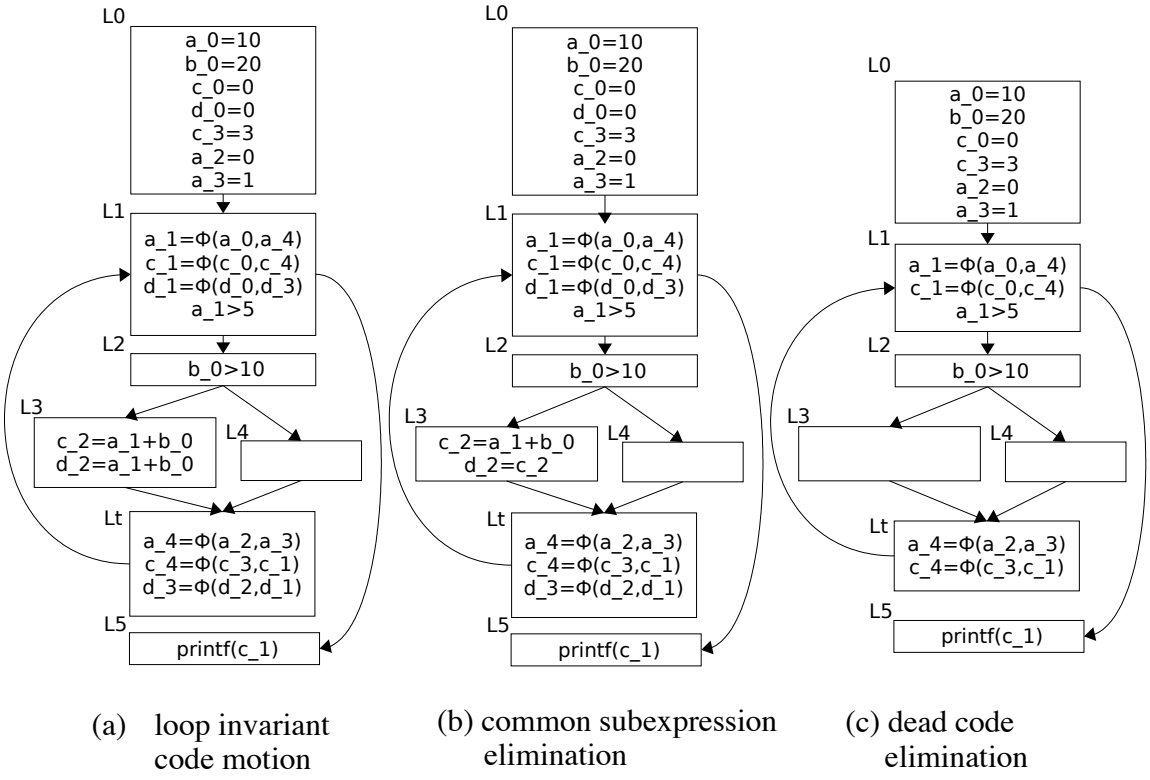

Figure 3: Examples of SSA optimization

We show conditional expressions can be checked in fine detail. In the intermediate code of COINS when a conditional expression contains logical operations && or ||, factors of the conditional expression are decomposed in advance into relational operations. For example, a statement

if(a&&(c < d)){

is converted to

if(a == true){
  if(c < d){

in the intermediate code. Therefore, each conditional expression in the intermediate code contains at most one relational operation. Our method makes comparison checks on the left and right-hand sides of relational operations of conditional expressions.

Parameters of function calls are another important place of "use". Therefore, we also perform comparison checking of parameters before and after function calls.

In summary, the following values are output as the trace:

- basic block number
- instruction number within the basic block
- name of the left-hand side variable and its value, in the case of an assignment statement
- symbol to indicate left- or right-hand side of a relational operation and its value, in the case of a conditional expression
- symbol to indicate parameter number and its value, in the case of a function call

```
__L1:

a_1__0 = ((int)( 0));
b_2__0 = ((int)( 0));
c_3__0 = ((int)( 0));
d_4__0 = ((int)( 0));
functionvalue_6__0 = ((int)( 0));
functionvalue_8__0 = ((int)( 0));
goto _L3;

_L3:
a_1__1 = ((int)( 10));
b_2__1 = ((int)( 20));
c_3__1 = ((int)( 0));
d_4__1 = ((int)( 0));
goto _L4;

_L4:
functionvalue_6__1 = phi(functionvalue_6__0:_L3,
functionvalue_6__2:_L8);
d_4__2 = phi(d_4__1:_L3, d_4__4:_L8);
c_3__2 = phi(c_3__1:_L3, c_3__5:_L8);
a_1__2 = phi(a_1__1:_L3, a_1__5:_L8);
if ((a_1__2 >  5)) { goto _L5;} else { goto _L9;}

_L5:
if ((b_2__1 >  10)) { goto _L6;} else { goto _L7;}

_L6:
c_3__3 = ((int)(((int)(a_1__2 + b_2__1))));
c_3__4 = ((int)( 3));
d_4__3 = ((int)(((int)(a_1__2 + b_2__1))));
a_1__4 = ((int)( 0));
goto _L8;

_L7:
a_1__3 = ((int)( 1));
goto _L8;

_L8:
d_4__4 = phi(d_4__3:_L6, d_4__2:_L7);
c_3__5 = phi(c_3__4:_L6, c_3__2:_L7);
a_1__5 = phi(a_1__4:_L6, a_1__3:_L7);
goto _L4;

_L9:
functionvalue_8__1 = printf((unsigned char *)&(string_9), c_3__2);
goto _L10;

_L10:
return;
```

Figure 4: Output of LirToC

- tag to indicate the entry block or the exit block of a loop

These are called "variable information".

3.2.7 SSA back translation and C-style program output

Because ϕ-functions in SSA form are hypothetical functions and cannot be executed, the SSA form is back-translated into normal form before code generation. This translation is called *SSA back translation*.

Before performing the SSA back translation, the system outputs a C language-style program, which is translated from the intermediate code in SSA form using a utility function (Java method) called LirToC in the backend module of COINS. Figure 4 is the actual C-style program output from Figure 2(d). We note that the control flow graphs shown in this section are based on the real program output by LirToC, with a small modifications to variable names and block numbers for readability.

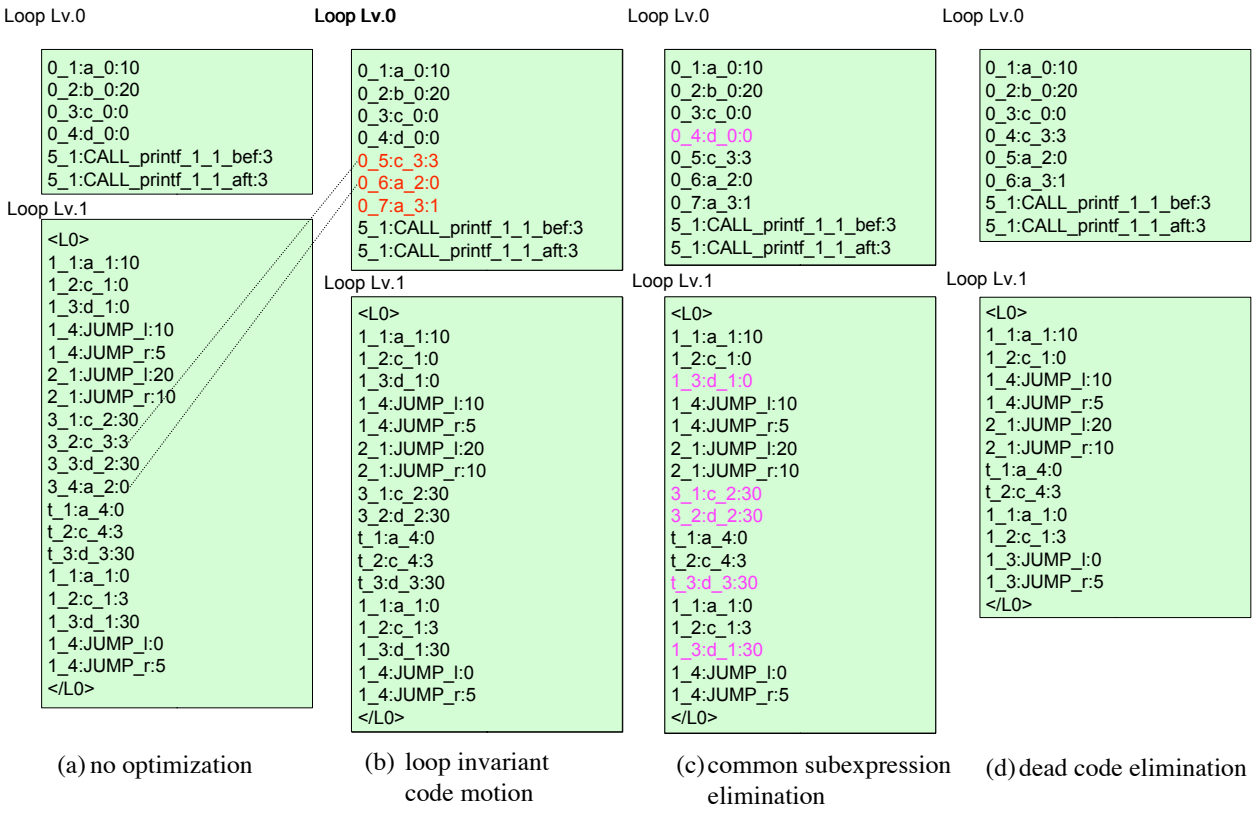

Figure 5: Variable information

Basic blocks and instruction numbers etc. in the trace information of the variables are made to correspond to the program output by LirToC. Therefore, when the comparison checker displays an error message, the optimizer writer can easily find the exact location of the bug in the intermediate code.

3.2.8 Execution

The system executes all object codes corresponding to each set of optimizations with the same input, and outputs the trace for each code. Figure 5 shows the trace output created by executing the programs shown in Figures 2(d) and 3.

Each line of "variable information" in the trace consists of several elements. The first element represents "*basic block number_instruction number in the basic block*". The second element represents the variable name etc., and the third element its value. Information in parentheses < and > represents the entry and exit of loops. Those with "/" are exits, and those without "/" are entries.

3.2.9 Comparison checking

Comparison checking is performed based on the output "variable information" in the trace. The comparison checking by Jaramillo et al. involves three phases: mapping, annotation, and comparison checking. However, in the SSA optimization the mapping and annotation phases can be omitted, using the characteristic feature of the SSA form that the definitions of the variables are textually unique. Therefore, comparison checking can be usually achieved by

simply comparing the values of the same variable name in the "variable information" before and after optimization.

The basic method is to compare each pair of the corresponding basic blocks before and after optimization. Therefore, the basic steps of comparison are as follows.

- First, save the "variable information" of the trace of a basic block before optimization in the "value pool".
- Then, look at each variable in the "variable information" of the corresponding basic block after optimization sequentially and compare its value with the value of the same variable from the "value pool".

However, for optimizations that make code motion or change the loop structure we cannot check by simply comparing the corresponding basic blocks. The handling of code motion etc. is described in the following subsections. For optimizations that change the loop structure we have also developed a comparison checking method [5, 6], but the description is omitted due to space limitations.

There are also variables, such as global variables and arrays, that are not translated into SSA form. Because they are not subject to SSA optimization, it is sufficient to compare these in the order of their appearance.

The method of comparison checking differs slightly with the type of optimization. They are described in the following subsections.

Instructions deleted or added by optimization

There are cases where instructions are deleted or added by optimization as in dead code elimination and operator strength reduction of induction variables. In these cases we cannot directly seek correspondence between variables before and after optimization. Therefore,

- We do not compare variables deleted or added by optimization. We compare only those variables that retain correspondence.

For example, in comparing Figure 5(c) and (d), variables `d_0, d_1, d_2, d_3` in Figure 5(c) are deleted by dead code elimination so these variables are not compared.

Operator strength reduction of induction variables and test replacement

If test replacement after operator strength reduction of an induction variable is performed the induction variable is changed. In this case, the values of both sides of relational operation of the replaced test do not coincide before and after optimization. To treat this case, we extended the optimizer so that it outputs

- basic block number for which test replacement is applied
- information of operator strength reduction of the induction variable for which the test replacement is applied

to a separate file. We performed a comparison using the above information in the file as a check.

Optimization involving code motion

In optimization for loop invariant code motion and partial redundancy elimination code is moved. In this case the correspondence relation cannot be established between instructions in the corresponding basic blocks before and after optimization. Therefore,

- the system compares instructions moved out of the loop after optimization with all instructions in the corresponding loop before optimization, by looking at the tags made by the loop analysis (subsection 3.2.5).

Therefore, it can naturally handle hoisting of loop invariants. This can be performed without additional mapping information from the optimizer.

For example, in comparing Figure 5(a) and (b), variables `c_3, a_2, a_3` are hoisted out from the loop in Figure 5(b) by loop invariant code motion. They are compared with the values of `c_3, a_2, a_3` in the loop of Figure 5(a), respectively. (In this example, `a_3` does not appear in Figure 5(a) because the relevant path was not executed.)

On the other hand, in the case of *partial dead code* elimination, the subject instruction moves down out of the loop, and this corresponds to the instruction that is computed the last time in the original loop. Therefore,

- as for instruction that is moved down out of the loop by optimization, the system compares it with the corresponding instruction that is computed the last time in the original loop, using the tags made by the loop analysis.

4 Experiments

4.1 Implementation

We used the COINS C compiler [3] (simply COINS in the following) to test its SSA optimizers [4], which have been under development. COINS is written in Java, so we also implemented our system in Java. We have added processing for comparison checking in COINS. Approximately 5000 lines were added, of which 2000 lines are for comparison checker.

4.2 Experiments

4.2.1 Purpose and criteria for evaluation

We experimented to test the correctness of the SSA optimizers in COINS using various test programs. Compared to Jaramillo et al.'s experiments [2], which use Lcc as the compiler, our experiments can be characterized by: use of a real project, checking each optimization pass separately and displaying C-style intermediate code to pinpoint the location of bugs.

In the experiments, we used the newest version at the time when we started the implementation: CVS version "July-10-2004" of COINS. It was used as the base compiler for all passes except the SSA optimizers.

For the SSA optimizers, branch versions and a some older versions were used to test the applicability of our method because the SSA optimizers in COINS were almost bug free at the time we started the experiments. SSA optimizers, except the common subexpression elimination based on question propagation (CSEQP), are taken from the release version "coins-0.10.1 (Mar-1-2004)". The CSEQP section is taken from the newest CVS version "July-10-2004". Partial redundancy elimination is taken from Tachikawa's implementation [7].

The optimizers used experimentally were: copy propagation, constant propagation, common subexpression elimination, CSEQP, partial redundancy elimination, loop invariant code motion, strength reduction and test replacement of induction variables, dead code elimination, removal of redundant ϕ-functions, and removal of empty basic blocks.

The test programs used were approximately 700 small test programs designed for testing COINS and 181.mcf from SPEC CINT2000.

4.3 Results

As the result of the experiments using the above test programs, we found four bugs in the COINS' SSA optimizers. The bugs of the optimizers found were as follows and the first two were unknown bugs:

- two for partial redundancy elimination
- one for common subexpression elimination based on question propagation
- one for operator strength reduction and test replacement of induction variable

By examining the error message and by comparing the two C-style programs generated from the SSA forms before and after optimization, it was relatively easy to find which part of the optimizer caused the bugs. Due to space limitation, we only present two of these, for the other bugs, see [5, 6].

4.3.1 Bugs in partial redundancy elimination

By applying our method, we found two bugs in the implementation of partial redundancy elimination (PRE) developed by Tachikawa [7]. We describe one of these.

This bug was assumed to be a bug in the dataflow equation for performing the PRE. Figure 6(a) is the program for which a bug was found. Figure 6(b) is the C-style program output by our method from the intermediate code (in SSA form) before applying PRE. A control flow graph is shown for readability. Figure 6(c) is the C-style program after applying PRE to the program of Figure 6(b). When we applied our method to the program in Figure 6(a), the system displayed an error message that the values of `c_2` of the second instruction of basic block L2 do not coincide before and after optimization.

In this case, PRE was applied to the basic blocks L0 and L2. When control returns from L5 to L1, the value of `a_1` at L1 must change and then the value of `c_2` at L2 in Figure 6(b) must also change, but the value of `temp_1` in Figure 6(c) was not updated. As a result, PRE was not performed correctly. We can infer that the computation of the dataflow equation for PRE is probably incorrect.

4.3.2 Bug in common subexpression elimination based on question propagation

By applying our method, we found a bug in the implementation of common subexpression elimination based on question propagation (CSEQP) in the COINS' SSA optimizer. The bug found by our method was that when CSEQP was applied after translating the SSA form intermediate code into three-address code, the semantics of the program was not preserved.

The program for which a bug was found is shown in Figure 7(a). Here, variables `a, b, c, x` are all global variables. If a global variable is used in some instruction, the pass for translation into three-address code first inserts an instruction that assigns that global variable to a local variable. Then, it translates the instruction into three-address code. Figure 7(b)

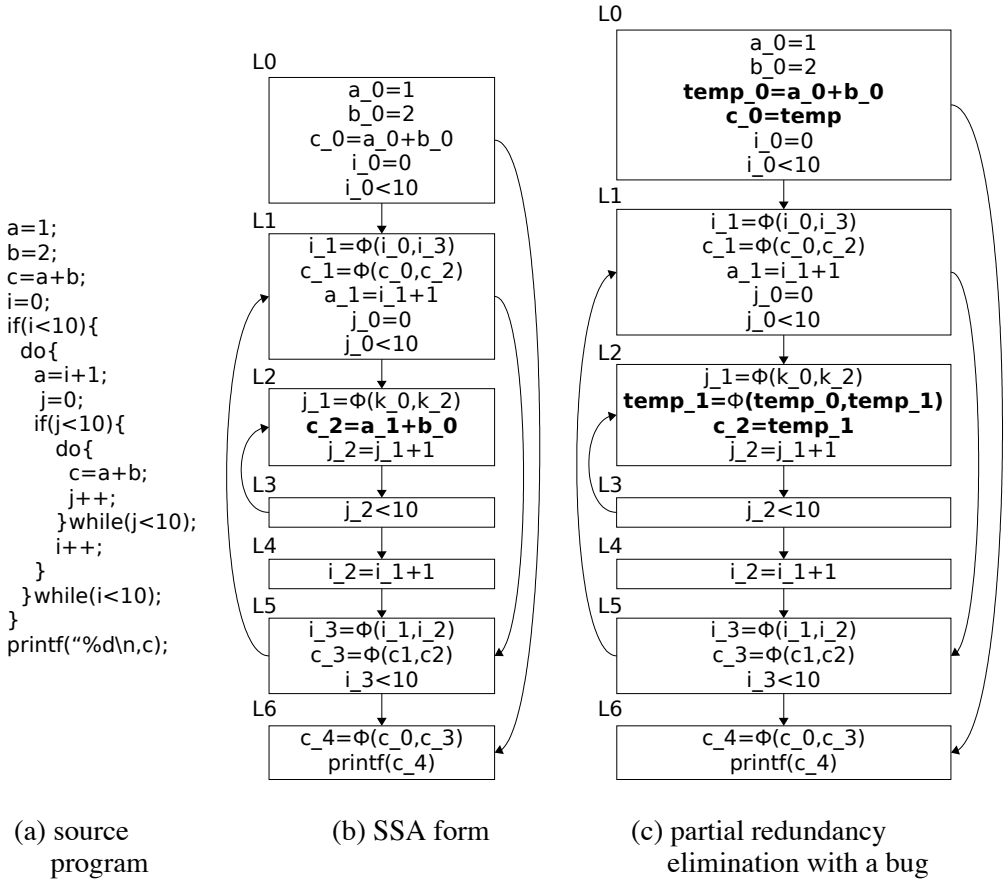

Figure 6: A bug in partial redundancy elimination

is the C-style program generated by applying our method, which shows the intermediate code after translating the intermediate code into three-address code. Figure 7(c) shows the C-style program after applying CSEQP to Figure (b). The statements in boldface in Figure 7(b) and (c) is the section where CSEQP was applied.

When our method was applied to this program, an error message "there are errors in `t_7, t_8, t_10, x`" was displayed. The reason for this error is that although the value of `t_7` in Figure 7(c) should have the value of `c` assigned by "`c=t_1+t_4`", its value is actually that of `t_3`, that is, an old value of `c`. From this, we can assume that this bug occurred because the optimizer ignored the condition that "global variables should not be treated as SSA form" and applied optimization to the global variables.

This bug occurs only when there are global variables in the program *and* translation into three-address code is required.

4.3.3 False alarm

False alarm is a situation where comparison checking displays errors although the optimization is correct.

Our method compares values before and after optimization. Among those values are values representing addresses that are dynamically allocated at runtime. Because these values of addresses are defined dynamically at runtime depending on the runtime environment, they may change if optimizations are applied. Therefore, values of address may cause false alarms.

Investigation of whether error messages displayed by our method were true or false showed

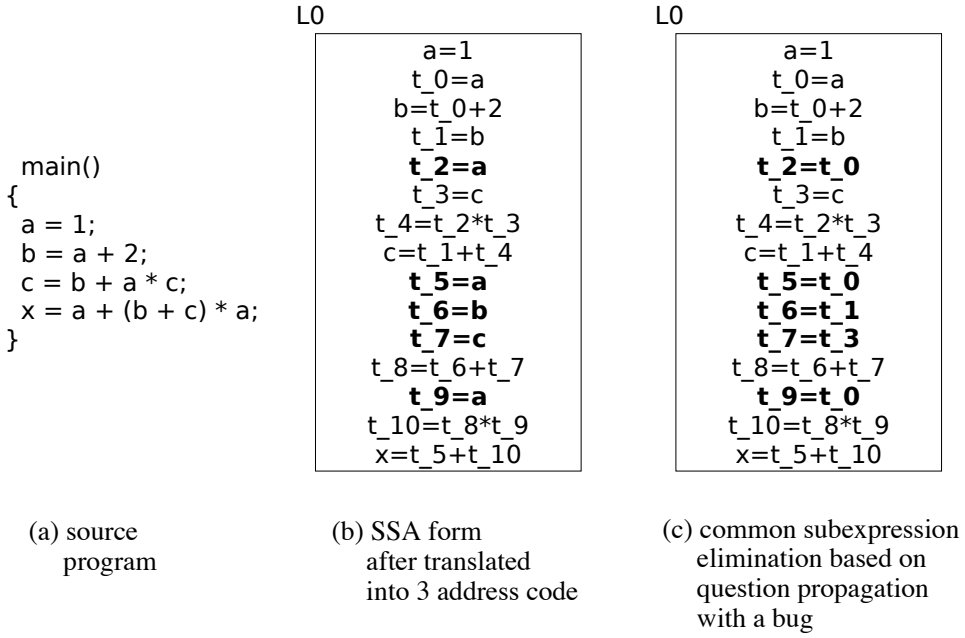

Figure 7: A bug in common subexpression elimination based on question propagation

that errors concerning source level integers usually have a small number of digits. Therefore, in our implementation, we provided an option to let the user specify the limiting number of digits for checking integers. In cases where the integer values do not match, integers greater than the specified number of digits are all assumed to be addresses, and no error messages are displayed.

By specifying this limit as seven digits, the ratio of *false alarm* becomes approximately 5% for the programs used in the experiments. In practice, values of addresses that are a *false alarm* can be easily recognized quickly.

5 Related Work

There are several studies of testing compiler optimization, such as those by Necula and Pnueli that make static comparisons but we omit the details as they are given in Jaramillo et al.'s paper.

Our method is based on Jaramillo et al.'s work [2], so it is very similar to theirs. However, we think that describing the issues met in the implementation and presenting the experiments and example bugs found in a real compiler project will be helpful to future practitioners.

6 Conclusions

In this paper, we describe our experience in testing compiler optimizers using comparison checking. Experiments using the COINS' C compiler are presented. As a result, we found four bugs in compiler optimizers including two unknown bugs. The *false alarm* in testing can be made as low as approximately 5%.

Although our method cannot prove the correctness of optimization, it is experimentally shown to be quite beneficial in practice because we can test optimizers to a relatively high level

of reliability.

Several issues are left to be followed up in the future.

The algorithm for applying our method to optimizations that change the loop structure is developed but no implementation has yet been made. Its implementation and experiments are planned.

The current implementation is based on an old version of the COINS compiler in which only a few SPEC benchmarks can be run, although it was the newest version when we started the implementation. Therefore, we plan to port our method on the newest version of the COINS compiler, and experimentally evaluate the method using large scale test programs and benchmarks.

Acknowledgments

This research was partially supported by the Japanese Ministry of Education, Culture, Sports, Science and Technology under the Grant "Special Coordination Fund for Promoting Science and Technology" and by the Japan Society for the Promotion of Science under the Grant-in-Aid for Scientific Research, and by the Kayamori Foundation for Informational Science Advancement.

References

[1] 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 Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 451–490 (1991).

[2] Jaramillo, C., Gupta, R. and Soffa, M. L.: Debugging and Testing Optimizers through Comparison Checking, *Electronic Notes in Theoretical Computer Science*, Vol. 65, No. 2, pp. 1–17 (2002).

[3] The Coins Project. Research of a common infrastructure for parallelizing compilers. http://www.coins-project.org/.

[4] Sassa, M., Nakaya, T., Kohama, M., Fukuoka, T. and Takahashi, M.: Static Single Assignment Form in the COINS Compiler Infrastructure, *Proc. SSGRR 2003w International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine on the Internet* (2003).

[5] Sudo, D. and Sassa, M.: Validation of compiler optimizers through comparison checking (in Japanese), *Proc. 7th JSSST Workshop on Programming and Programming Languages (PPL2005)*, pp. 231–245 (2005).

[6] Sudo, D.: Validation of compiler optimizers through comparison checking (in Japanese), Master Thesis, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology (2005).

[7] Tachikawa, S.: Partial Redundancy Elimination on SSA Form (in Japanese), Master Thesis, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology (2004).