Electronic Notes in Theoretical Computer Science 65 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 17 pages

Debugging and Testing Optimizers through Comparison Checking

Clara Jaramillo

Dept. of Computer Science
Chatam College
Pittsburgh, USA

Rajiv Gupta

Dept. of Computer Science
University of Arizona
Tucson, USA

Mary Lou Soffa

Dept. of Computer Science
University of Pittsburgh
Pittsburgh, USA

Abstract

We present a novel technique called comparison checking that helps optimizer writers debug optimizers by testing, for given inputs, that the semantics of a program are not changed by the application of optimizations. We have successfully applied comparison checking to a large class of program transformations that alter (1) the relative ordering in which values are computed by the intermediate code statements, (2) the form of the intermediate code statements, and (3) the control flow structure using code replication. We outline the key steps that lead to the automation of comparison checking. The application of comparison checking to test the implementations of high level loop transformations, low level code optimizations, and global register allocation for given program inputs is then described.

1 Introduction

As compilers increasingly rely on optimizations to achieve high performance, the technology to debug optimized code continues to falter. The problem of debugging optimized code is two fold because errors in an optimized program can originate in the source program or be introduced by the optimizer. Therefore tools must be developed to help application programs debug optimized

code and optimizer writers debug optimizers. In this paper, we focus on a tool developed for the optimizer writer. In particular, we present a novel technique called comparison checking that helps optimizer writers debug optimizers by testing, for given inputs, that the semantics of a program are not changed by the application of optimizations.

The comparison checker, as illustrated in Figure 1, automatically orchestrates the executions of both the unoptimized and optimized versions of a source program, for given inputs, and compares their semantic behaviors. The semantic behavior of an unoptimized or optimized program with respect to the source program is characterized by the outputs and values computed by source level statements in the unoptimized or optimized program for all possible inputs. Therefore, the semantic behaviors of the unoptimized and optimized programs with respect to the source program are compared by checking that (1) the same paths are executed in both programs, (2) corresponding source level assignments compute the same values and reference (i.e., read, write) the corresponding locations, and (3) the outputs are the same. The outputs and the values computed by source level assignment statements and branch predicates for given inputs are compared in both versions. In addition, for assignments through arrays and pointers, checking is done to ensure the addresses to which the values are assigned correspond to each other. All assignments to source level variables are compared with the exception of those dead values that are not computed in the optimized code.

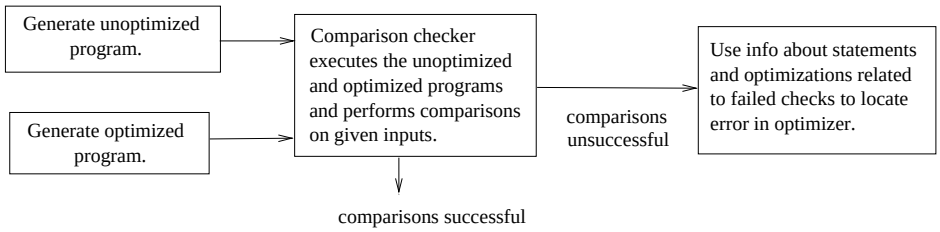


Fig. 1. The comparison checking system.

If the semantic behaviors are the same and correct with respect to the source program, the optimized program can be run with high confidence. On the other hand, if the semantic behaviors differ, the comparison checker displays the statements responsible for the differences and the optimizations applied to these statements. Our approach to checking allows the system to locate the *earliest* point where the unoptimized and optimized programs differ in their semantic behavior with respect to the source program. For example, the checker detects the earliest point during execution when corresponding source level statement instances should but do not compute the same values. Therefore, the checker can detect statements that are incorrectly optimized and subsequently compute incorrect values. The optimizer writer can use this information to locate the incorrect code in the optimized program and determine what transformation(s) produced the incorrect code.

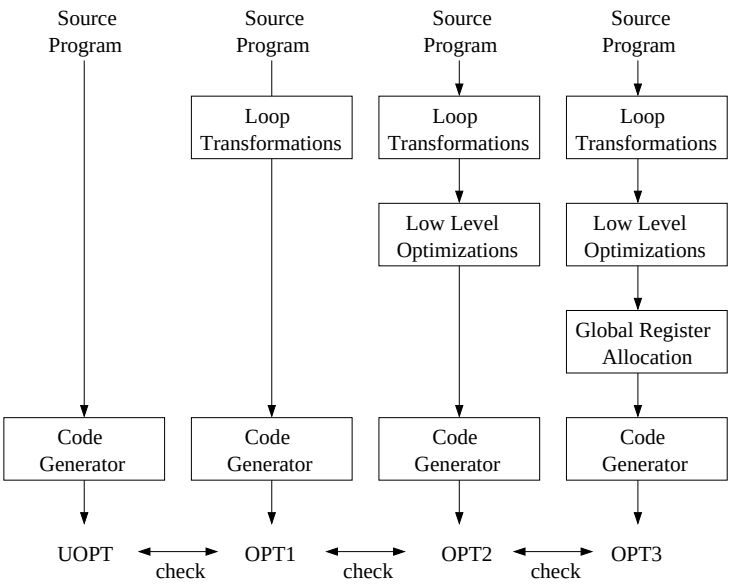


Fig. 2. Phased comparison checking.

The application of comparison checking can be performed in several phases. For example, as illustrated in Figure 2, the optimization task can be divided into three phases: high level transformation, low level optimizations, and global register allocation. In each comparison checking step, a pair of program versions is compared to determine whether a given optimization phase preserves the program's semantics. If comparison checking fails, this approach focuses the attention of the optimizer writer on the part that contains a bug. If we simply compare the unoptimized and fully optimized programs, and comparison checking fails, the optimizer writer must consider all three parts of the optimizer as possible sources of error.

The remainder of this paper is organized as follows. In section 2 we describe the comparison checking scheme that handles loop transformations as well as low level code optimizations. In section 3 we present an overview of our comparison checking scheme for global register allocation. In section 4 we present some experimental results. We comparing our approach with related work in section 5 and conclude in section 6.

2 Loop Transformations and Code Optimizations

Our approach handles code optimized using a large class of code optimizations that can be characterized in terms of their effect on the program code as follows. The transformations may alter the relative ordering in which values are computed by the intermediate code statements (e.g., code motion based partial redundancy and partial dead code elimination), the form of the intermediate code statements (e.g., constant propagation and reassociation), and the control flow structure using code replication (e.g., function inlining and

loop unrolling).

To automate comparison checking we must (1) determine which values computed by both programs need to be compared with each other, (2) determine where the comparisons are to be performed in the program executions, and (3) perform the comparisons. To achieve these tasks, the following three sources of information are utilized:

(1) Mappings. To compare values computed in both the unoptimized and optimized programs, we need to determine the corresponding statement instances that should compute the same values. A statement instance in the unoptimized program and a statement instance in the optimized program are said to *correspond* if the values computed by the two instances should be the same and the latter was derived from the former by the application of some optimizations. Our mappings associate *statement instances* in the unoptimized program and the corresponding *statement instances* in the optimized program. In [8], we developed a mapping technique to identify corresponding statement **instances** and describe how to generate mappings that support code optimized with classical optimizations as well as loop transformations.

(2) Annotations. The mappings are used to automatically generate *annotations* for the unoptimized and optimized programs, which guide the comparison checker in comparing corresponding values and addresses. When a program point in either program version that has annotations is reached, the actions associated with the annotations at that point are executed by the comparison checker. Annotations identify program points where comparison checks or other activities should be performed to enable checking. Breakpoints are introduced to extract values and activate annotations associated with program points in the unoptimized and optimized programs.

(3) Value Pool. Since values to be compared are not always computed in the same order in the unoptimized and optimized code, a mechanism saves values that are computed early. These values are saved in a *value pool* and removed when no longer needed. Annotations are used to indicate if values should be saved in the value pool or discarded from the value pool.

Comparison Checking Algorithm

The execution of the unoptimized program drives the checking and the execution of the optimized program. Therefore, execution begins in the unoptimized code and proceeds until a breakpoint is reached. Using the annotations in the unoptimized code, the checker determines if the value computed can be checked at this point. A breakpoint at a program point in the unoptimized code that has a *comparison check* annotation indicates that a value computed by some statement should be checked. Also, by default, a breakpoint at a program point in the unoptimized code that has no associated annotation indicates that the value computed by the most recently executed statement should be checked. If a value should be checked, the optimized program executes until the corresponding value is computed (as indicated by a comparison

check annotation), at which time the check is performed on the two values. During the execution of the optimized program, any values that are computed "early" (i.e., the corresponding value in the unoptimized code has not been computed yet) are saved in the value pool, as directed by the *save* annotations. If a *delay comparison check* annotation is encountered, indicating the checking of the value computed by the unoptimized program cannot be performed at the current point, the value is saved for future checking. The checker continues to alternate between executions of the unoptimized and optimized programs. Annotations also indicate when values that were saved for future checking can finally be checked and when the values can be removed from the value pool. Values computed by statement instances that are deleted by an optimization are not checked.

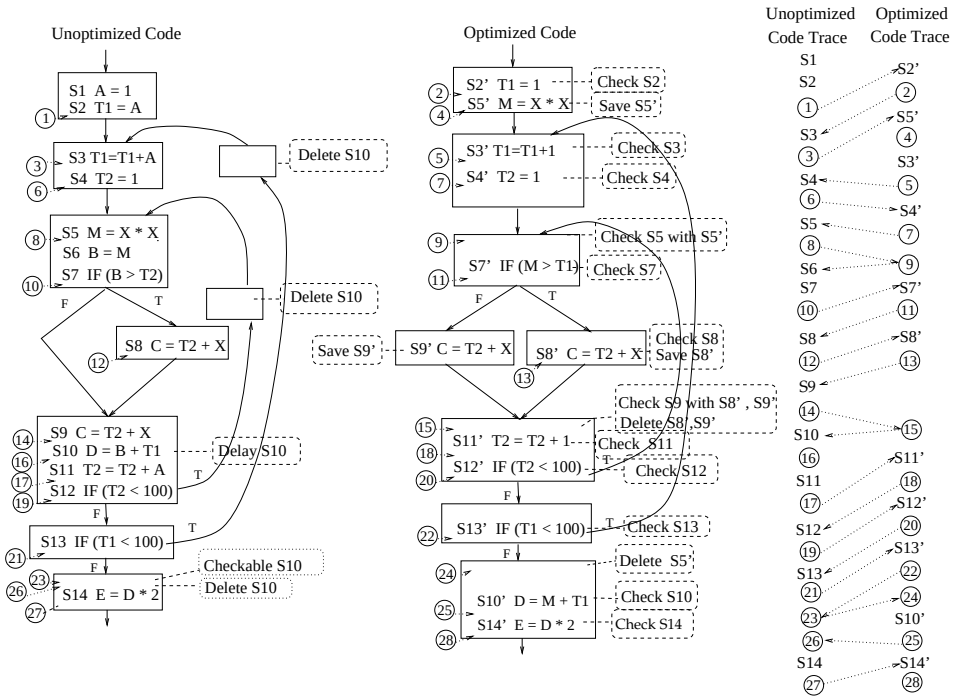
One of the attractive features of comparison checking is that it does not report false positives. This is because it relies upon precise dynamic information. It may appear that the optimizer can be tested by running the unoptimized and optimized program versions and comparing their outputs. However, in a given run, a typical program is expected to compute many intermediate results that are eventually discarded. Therefore while comparison checking will detect errors in computing these intermediate results, a simple comparison of unoptimized and optimized program outputs will fail to detect the same errors.

An Example

In Figure 3(a), the unoptimized program and the optimized program version as well as the annotations are shown. For ease of understanding, the source level statements shown are simple. The following optimizations were applied.

- constant propagation - the constant 1 in $S1$ is propagated, as shown in $S2'$, $S3'$, and $S11'$.
- copy propagation - the copy M in $S6$ is propagated, as shown by $S7'$ and $S10'$.
- dead code elimination - $S1$ and $S6$ are dead after constant and copy propagation.
- loop invariant code motion - $S5$ is moved out of the doubly nested loop. $S10$ is moved out of the inner loop.
- partial redundancy elimination - $S9$ is partially redundant with $S8$.
- partial dead code elimination - $S10$ is moved out of the outer loop.

Assume all the statements shown are source level statements and loops execute for a single iteration. Breakpoints are indicated by circles. Breakpoints are placed at program points in the unoptimized and optimized code that are associated with annotations as well as program points in the unoptimized code where comparison checks should be performed. Given the annotations (dis-



(a) Annotated unoptimized and optimized code (b) Check traces

Fig. 3. Comparison checking scheme example.

played in dotted boxes), we now describe the operations of the checker on this example. The switching between the unoptimized and optimized program executions by the checker is illustrated by the traces in Figure 3(b). The traces include the statements executed as well as the breakpoints (circled) where annotations are processed. The arrows indicate the switching between programs.

The annotation *Check S with S'* indicates that it is now time to compare the value computed statement S in the unoptimized code with the corresponding value computed by statement *S'* in the optimized code. If the annotation is placed immediately following statement *S'* in the optimized code, then the *with S'* portion of the annotation is omitted. The annotation *Save* is used to save a computed value in the *value pool* while the annotation *Delete* is used to discard a value from the value pool as all checks involving the value have already been performed. A pair of *Delay* and *Checkable* annotations are used in tandem by the unoptimized code. The *Delay* annotation indicates that a value cannot be checked at this point because the optimized code could not have as yet computed the value while the *Checkable* annotation indicates that it should now be possible to check the value.

The unoptimized program starts to execute with $S1$ and continues executing without checking $S1$, as it was deleted from the optimized program. After $S2$ executes, breakpoint 1 is reached and the value computed at $S2$ can be checked at this point and so the optimized program executes until *Check* $S2$ is processed, which occurs at breakpoint 2. The values computed by $S2$

and $S2'$ are compared. The unoptimized program resumes execution and the loop iteration at $S3$ begins. After $S3$ executes, breakpoint 3 is reached and the optimized program executes until *Check* $S3$ is processed. Since a number of comparisons have to be performed using the value computed by $S5'$, when breakpoint 4 is reached, the annotation *Save* $S5'$ is processed and consequently, the value computed by $S5'$ is stored in the value pool. The optimized code continues executing until breakpoint 5, at which time the annotation *Check* $S3$ is processed. The values computed by $S3$ and $S3'$ are compared. $S4$ then executes and its value is checked. $S5$ then executes and breakpoint 8 is encountered. The optimized program executes until the value computed by $S5$ can be compared, indicated by the annotation *Check* $S5$ *with* $S5'$ at breakpoint 9. The value of $S5$ saved in the value pool is used for the check. The programs continue executing in a similar manner.

3 Global Register Allocation

In this section, a register allocation checker is developed that can detect errors in a register allocator implementation and determine the possible cause(s) of the errors. This checker saves different kinds of information and utilizes a different set of annotations to track information about the variables that are assigned to registers and verify that the expected values of these variables are used throughout the optimized program execution. When a register allocation technique is implemented incorrectly, the incorrect behavior can include:

- using a wrong register,
- overwriting a live register,
- evicting a value from a register but not saving it for future uses,
- failing to load a value from memory, and
- using a *stale* value. A stale value of a variable is used in the optimized program when a value of a variable is computed in a register, but instead of using the value of the variable in the register, the optimized program uses the old value in memory.

The register allocation checker can determine when a register allocator exhibits these types of behavior. Consider the unoptimized C program fragment and its optimized version in Figure 4. Assume the unoptimized program is correct and the register allocation is turned on. The register allocator assigns variables x, y, z, and a to registers $r3$, $r1$, $r4$, and $r5$, respectively, in the optimized program, and copies the value of y in register $r1$ to register $r2$. Assume the optimized program returns incorrect output. Using the checker, a difference is detected in the internal behavior of the unoptimized and optimized programs at line 3 in the unoptimized code and line 7 in the optimized code. The checker indicates that the values used by y in the unoptimized and optimized programs differ and indicates that $r1$ is used inconsistently as y was

evicted from $r1$ earlier during the execution of the optimized program. The checker also indicates that the expected value of y is in register $r2$ and thus, $r2$ should have been used instead of $r1$. The optimizer writer can then use this information to debug the implementation of register allocation.

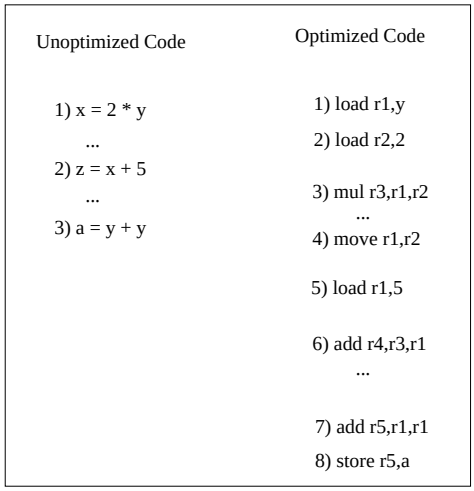


Fig. 4. Program example for register allocation checking.

The register allocation checking scheme is similar to the comparison checking scheme in that values computed in both the unoptimized and optimized programs are checked, but the register allocation checking scheme also compares values of variables that are used in both programs and tracks information about the variables that are assigned to registers throughout the optimized program execution. This tracking information includes maintaining at each program point of the execution of the optimized program the

(1) current locations of values of variables,

(2) variables whose memory locations hold stale values,

(3) variables whose values in registers have been evicted, and

(4) variables that are currently assigned to registers.

To achieve these tasks, again mappings and annotations are utilized. The mappings of the comparison checker are extended to include the correspondences between the *uses of variables* in the unoptimized and optimized intermediate programs. These mappings are generated before register allocation is applied because the correspondence between the two program versions is not changed by the application of register allocation. The mappings capture only the effects of register allocation and not other optimizations because the checking is performed on a program before registers are allocated and on a program after registers are allocated. However, the unoptimized program can include the application of other optimizations, which are assumed to be correct. After register allocation is applied and code is generated by the compiler, the mappings are used to automatically generate *annotations* for the

optimized program, which guide the register allocation checker in comparing corresponding values and addresses and tracking information about the variables that are assigned to registers throughout the program execution. When a targeted program point in the optimized code is reached, the actions associated with the annotations at that point are executed by the register allocation checker.

A high level conceptual overview of the register allocation checker algorithm is given in Figure 5. This algorithm is similar to that of the comparison checker. Breakpoints are used to extract values from the unoptimized and optimized programs as well as activate annotations. Annotations guide the actions of the register allocation checker. However, since values that are assigned to variables in the optimized code are the values that are tracked and checked, the execution of the optimized program drives the checking and the execution of the unoptimized program. Therefore, execution begins in the optimized code and proceeds until a breakpoint is reached. Depending on the annotation, the checker may track variables assigned to registers, evicted variables, and stale variables, and/or determine if a value computed or used should be checked at the current point of execution of the optimized code. When a value should be checked, the unoptimized program executes until the corresponding point of execution is reached, at which time the check is performed on the two values. The checker continues to alternate between executions of the unoptimized and optimized programs. If the values that are compared differ, then the checker informs the user of the possible causes of the difference. Also, as values are tracked, the checker informs the user of any inconsistencies (e.g., a stale value is loaded, unexpected value is stored to a memory location, etc.). Once an inconsistency of a value of a variable is detected, the inconsistency is propagated through the uses of the value.

```
do
  Execute the optimized program and process annotations at breakpoint
  if Check annotation then
        Execute the unoptimized program until the equivalent execution
        point (of the optimized program) is reached
        Perform the comparison check
        if error then report the error and the cause of the error
  if Register Assign annotation or Load annotation or Store annotation
     or Register Move annotation then
        Update register/variable information
        Verify the loaded or stored value is the expected value
        Inform user of any inconsistencies
  endif
while the optimized program has not finished executing
```

Fig. 5. Register allocation checker algorithm.

Annotations

Similar to the comparison checking technique, code annotations guide the checking of values in the unoptimized and optimized code. Code annotations are also used to track values of variables. Annotations (1) identify program points where comparison checks should be performed and (2) indicate what values of variables/registers should be tracked and checked in the optimized code. Five types of annotations are needed to implement the register allocation checking strategy. In the example in Figure 6, which illustrates the same unoptimized and optimized code example given in Figure 4, annotations are shown in dotted boxes.

(1) The Check $\mathbf{v}, \mathbf{r}$ *annotation.* The *check* v, r annotation is associated with a program point p in the optimized code to indicate a check of a value of variable v in register r is to be performed. The register allocation checker will execute the unoptimized program until the equivalent program point p is reached. The corresponding value to be compared is the current value of v in the optimized code. For example, in Figure 6, a *check* annotation is associated with statement $S1'$ in the optimized code so that the contents of $r1$ in the optimized code is compared with the value of y in the unoptimized code. *Check* annotations are used to check register loads, stores, uses, and assignments.

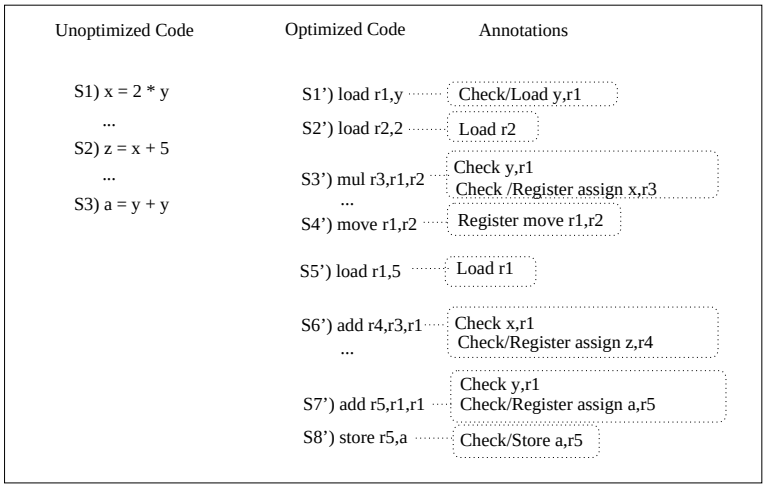


Fig. 6. Annotations example.

(2) The Register assign $[\mathbf{v},]$ $\mathbf{r}$ *annotation.* The *register assign* annotation is associated with a program point in the optimized code to indicate the tracking information for register r should be updated. The register allocation checker records that the previous variable assigned to r is evicted. If v is specified, the register allocation checker updates its information to indicate that r holds variable v, v is currently stored in r, the memory location of v holds a stale value, and any other values of v currently in registers are evicted. For example, in Figure 6, a *register assign* annotation is associated with statement $S3'$ in the optimized code so that the variable x is tracked with register $r3$ in the

optimized code.
(3) The Load $[\mathbf{v},]$ $\mathbf{r}$ *annotation.* The *load* annotation is associated with a *load* instruction in the optimized code and is used to track the load information for register r. The register allocation checker records that the previous variable assigned to r is evicted. If v is specified, the register allocation checker records that r holds variable v, v is currently stored in r, and any other values of v currently in registers are evicted. Using the tracking information, the checker checks if the loaded value is stale, and if so, records this information, informs the user of the stale value of v, and informs the user of the current location of the expected value of v, if it exists. For example, in Figure 6, a *load* annotation is associated with statement $S1'$ in the optimized code to track and check the information in register $r1$.
(4) The Store $\mathbf{v}, \mathbf{r}$ *annotation.* The *store* annotation is associated with a *store* instruction in the optimized code to track the store information for register r. Using the tracking information, the checker checks if r does not hold the expected value of v, and if so, informs the user that r does not hold the expected value of v and informs the user of the current location of v, if it exists. Also, the register allocation checker records that the memory location of v holds the current value. For example, in Figure 6, a *store* annotation is associated with statement $S8'$.
(5) The Register move $\mathbf{r}, \mathbf{r'}$ *annotation.* The *register move* annotation is associated with a *move* instruction in the optimized code to track the information in register r'. The register allocation checker duplicates the information pertaining to register r for that of register r'. For example, in Figure 6, a *register move* annotation is associated with statement $S4'$ in the optimized code to track the information in register $r2$.

Combining Annotations

When a *Check* annotation is associated with a *Store* annotation, the checker checks that the value in the register stored in the memory location of the variable at a program point in the optimized program matches the value of the variable at the equivalent program point in the unoptimized program. If the values do not match, then if the register currently holds the variable, then the checker informs the user why the value in the optimized code is incorrect. Either the value is stale, uninitialized, or wrong (possibly because the correct value was evicted and now the register contains the wrong value that will be stored). If the register does not currently hold the variable, the checker informs the user (1) if the expected value of the variable resides in another register, (2) the last location of the variable, and (3) that either the wrong register or address was supplied in the instruction, the expected value was evicted earlier and not saved, or the memory value already has the expected value (because of an earlier store). A *Check* annotation associated with a *Load* annotation is treated in a similar manner.

When a *Check* annotation is associated with a *Register assign* annotation,

the checker verifies that the value assigned to the register at a program point in the optimized code matches the value of the variable at the equivalent program point in the unoptimized code. If the operands were incorrect, the checker will have already notified the user of the uses that have unexpected values. Otherwise, incorrect code was generated.

Annotation Placement

Annotations are placed in the optimized program as follows. Using the mappings, *Check/Register assign* annotations are placed on every variable assignment in the optimized code and *Check* annotations are placed on every variable use in the optimized code. Next, at every instruction in the optimized code that stores to a register, *Register assign* annotations are placed, except at the program points where *Check/Register assign* annotations have been placed. At every instruction in the optimized code that loads a variable into a register, *Check/Load* annotations are placed. At all other load instructions in the optimized code, *Load* annotations are placed. Similarly, at every instruction in the optimized code that stores to a memory location of a variable, *Check/Store* annotations are placed. At all other store instructions in the optimized code, *Store* annotations are placed. Finally, at every move instruction in the optimized code, *Register move* annotations are placed.

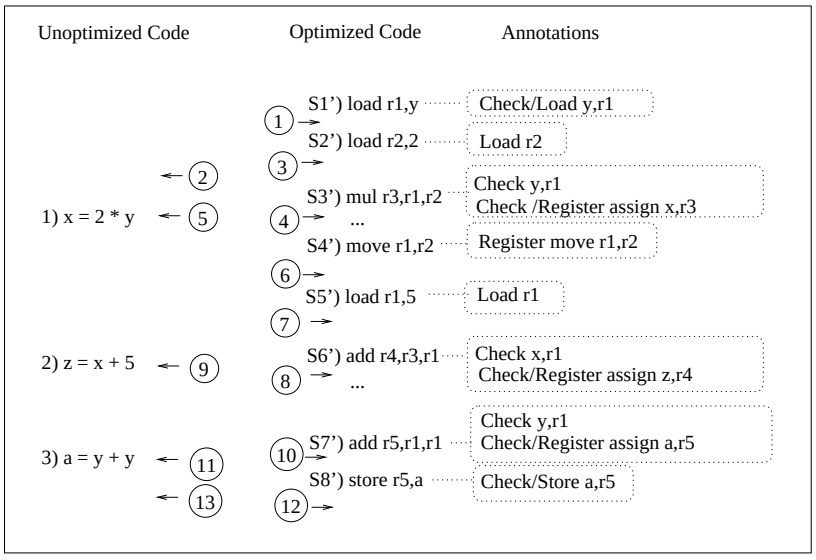


Fig. 7. Register allocation checker example.

An Example

Consider the annotated unoptimized and optimized program segments in Figure 7, which illustrate the same unoptimized and optimized code example given in Figure 4. Breakpoints are indicated by circles. Annotations are shown in dotted boxes. The optimized program starts to execute with $S1'$, and breakpoint 1 is reached. The checker determines from the annotation

that the value loaded into register $r1$ should be compared with the value of y in the unoptimized code at the equivalent program point in the unoptimized code. Thus, the unoptimized program executes until breakpoint 2 is reached, at which time the checker compares the value of y in the unoptimized program with the value of $r1$ in the optimized code. If the values are the same, the checker determines from the *Load* annotation that information regarding y and $r1$ should be tracked. The checker records that $r1$ now holds the value of y and that y is currently stored in $r1$. If y is stored in any other register, the checker records that y is evicted from these other registers. Also, if the loaded value is stale, the checker informs the user of the stale value and the location of the expected value of the variable (if it exists).

The optimized program continues execution and breakpoint 3 is reached. The checker processes the *Load* annotation by recording that the latest variable in $r2$ is now evicted. The optimized and unoptimized programs continue executing in a similar manner.

Notice that when breakpoint 6 is reached, checker processes the *Register move* annotation and records that $r2$ holds the value of y and y is stored in $r2$. At breakpoint 7, the checker processes the *Load* annotation and records that y is evicted from $r1$. At breakpoint 10, the checker processes the Check annotation by executing the unoptimized program until breakpoint 11 is reached, at which time the value of y in the unoptimized code is compared with $r1$. The values differ and the checker informs the user that y was evicted from $r1$ and the expected value of y in the optimized code is in $r2$.

4 Implementation and Experiments

The **C**omparison checker for **OP**timized code, COP, was implemented, including the instruction mapping, annotation placement, and checking. `Lcc` [7] was used as the compiler for an application program and was extended to include a set of optimizations, namely loop invariant code motion, dead code elimination, partial redundancy elimination, register allocation, copy propagation, and constant propagation and folding. On average, the optimized code generated by the optimized `lcc` executes 16% faster in execution time than the unoptimized code.

As a program is optimized, mappings are generated. Besides generating target code, `lcc` was extended to determine the mappings between the unoptimized and optimized code, breakpoint information, and annotations that are derived from the mappings. The code to emit breakpoint information and annotations is integrated within `lcc` through library routines. Thus, compilation and optimization of the application program produce the target code for both the unoptimized program and optimized program as well as auxiliary files containing breakpoint information and annotations for both the unoptimized and optimized programs. These auxiliary files are used by the checker.

Table 1
Execution times (minutes:seconds).

Program	Source length (lines)	Unoptimized Code (CPU)	Unoptimized Code annotated (CPU)	Optimized Code (CPU)	Optimized Code annotated (CPU)	COP (CPU)	COP (response time)
`wc`	338	00:00.26	00:02.16	00:00.18	00:01.86	00:30.29	00:53.33
`yacc`	59	00:01.10	00:06.38	00:00.98	00:05.84	01:06.95	01:34.33
`go`	28547	00:01.43	00:08.36	00:01.38	00:08.53	01:41.34	02:18.82
`m88ksim` [1]	17939	00:29.62	03:08.15	00:24.92	03:07.39	41:15.92	48:59.29
`compress` [1]	1438	00:00.20	00:02.91	00:00.17	00:02.89	00:52.09	01:22.82
`li` [1]	6916	01:00.25	05:42.39	00:55.15	05:32.32	99:51.17	123:37.67
`ijpeg` [1]	27848	00:22.53	02:35.22	00:20.72	02:33.98	38:32.45	57:30.74

[1] Spec95 benchmark test input set was used.

Breakpoints are generated whenever the value of a source level assignment or a predicate is computed and whenever array and pointer addresses are computed. Breakpoints are also generated to save base addresses for dynamically allocated storage of structures (e.g., `malloc()`, `free()`, etc.). Array addresses and pointer addresses are compared by actually comparing their offsets from the closest base addresses collected by the checker. Floating point numbers are compared by allowing for inexact equality. That is, two floating point numbers are allowed to differ by a certain small delta [10]. Breakpointing is implemented using fast breakpoints [9].

Experiments were performed to assess the practicality of COP. The main concerns were usefulness as well as cost of the comparison checking scheme. COP was found to be very useful in actually debugging the optimizer implemented for this work. Errors were easily detected and located in the implementation of the optimizations as well as in the mappings and annotations. When an unsuccessful comparison between two values was detected, COP indicated which source level statement computed the value, the optimizations applied to the statement, and which statements in the unoptimized and optimized assembly code computed the values.

In terms of cost, the slow downs of the unoptimized and optimized programs and the speed of the comparison checker are of interest. COP performs *on-the-fly* checking during the execution of both programs. Both value and address comparisons are performed. In the experiments, COP ran on an HP 712/100 and the unoptimized and optimized programs on separate SPARC 5 workstations instead of running all three on the same processor as described in Section 3. Messages are passed through sockets on a 10 Mb network. A buffer is used to reduce the number of messages sent between the executing programs and the checker. Some of the integer Spec95 benchmarks as well as some smaller test programs were used as test cases.

Table 1 shows the CPU execution times of the unoptimized and optimized programs with and without annotations. On average, the annotations slowed down the execution of the unoptimized programs by a factor of 8 and that of the optimized programs by a factor of 9. The optimized program experiences

greater overhead than the unoptimized program because more annotations are added to the optimized program.

Table 1 also shows the CPU and response times of COP. The performance of COP depends greatly upon the lengths of the execution runs of the programs. Comparison checking took from a few minutes to a few hours in terms of CPU and response times. These times are clearly acceptable if comparison checking is performed off-line (i.e., non-interactively). The performance of the checker was found to be bounded by the processing platform and speed of the network. A faster processor and 100 Mb network would considerably lower these times. In fact, when COP executes on a 333 MHz Pentium Pro processor, the performance is on average 6 times faster in terms of CPU time. Access to a faster network was not possible. The pool size was measured during experiments and was fairly small. If addresses are not compared, the pool size contains less than 40 values for each of the programs. If addresses are compared, then the pool size contains less than 1900 values.

5 Related Work

Not much work has focused on developing tools to help debug optimizers. Bugfind [1] was developed to help debug optimizers by pinpointing which functions produce incorrect code. This tool also helps application writers by compiling each function to its highest level of correct optimization. To achieve these tasks, functions must be placed in separate files. Boyd and Whalley[2] developed the `vpoiso` tool to identify the first transformation during optimization that causes the output of the execution to be incorrect. A graphical optimization viewer, `xvpodb`, was developed and allows users to view the state of the generated instructions before and after each application of transformations. However, if the optimizer writer cannot conclude which specific instructions in the optimized code produce incorrect results, using `xvpodb` is tedious.

More recent work [3,4] statically compares the intermediate form of a program before and after a compilation pass and verifies the preservation of the semantics. This work symbolically evaluates the intermediate forms of the program and checks that the symbolic evaluations are equivalent. While this approach validates a program translation for all possible program inputs, it also detects false alarms that are not necessarily errors. In contrast, comparison checking does not detect false alarms but it only tests the translation of the program for the inputs on which the the unoptimized and optimized program versions are run.

Another approach for debugging optimizers is by simply using tools for debugging optimized code and let the user infer whether the error is in the source code or the optimizer. Not only does this approach makes the task of the user much more difficult; in addition, even the most recent works for debugging optimized code cannot fully report expected values of variables in

the unoptimized program [11,5,6].

6 Conclusion

In this paper, we presented comparison checking, a technique that is designed to help optimizers writers debug their optimizers by testing that the unoptimized and optimized versions of a program have the same semantics with respect to program inputs. The comparison checking technique uses mappings and annotations to guide the checking. The mappings are determined as optimizations are performed and thus require that the optimizer writer extend the optimization implementation to include mappings. The annotations and their placement are automatically determined from mappings. During checking, if the semantics of the two versions differ, the checker can pinpoint where the earliest difference occurs and the optimizations that were involved.

The approach of comparison checking is different from the traditional debugging approach in which the user queries the debugger. The checker works without any user queries. It is also different from the verification approach in that the checker, for a given input, uses precise dynamic information to compare the semantic behaviors. The verification techniques works for any input but generates false positives. No false positives are generated by the checker.

References

[1] Caron, J.M. and Darnell, P.A. Bugfind: A Tool for Debugging Optimizing Compilers. *Sigplan Notices*, 25(1):17–22, January 1990.

[2] Boyd, M.R. and Whalley, D.B. Isolation and Analysis of Optimization Errors. In *Proceedings ACM SIGPLAN'93 Conf. on Programming Languages Design and Implementation*, pages 26–35, June 1993.

[3] Necula, G. Translation Validation for an Optimizing Compiler. In *Proceedings ACM SIGPLAN'99 Conf. on Programming Languages Design and Implementation*, pages 83–94, June 2000.

[4] Pneuli, A., Rodeh, Y., Shtrichman, O., and Siegel, M. In *Proceedings 11th International Conference on Computer Aided Verification*, LNCS 1633, Springer-Verlag, pages 455-469, 1999.

[5] Wu, L. *Interactive Source-Level Debugging of Optimized Code.* PhD dissertation, University of Illinois, Urbana-Champaign, 2000.

[6] Dhamdhere, D. M. and Sankaranarayanan, K. V. Dynamic Currency Determination in Optimized Programs. *ACM Transactions on Programming Languages and Systems*, 20(6):1111–1130, November 1998.

[7] Fraser, C. and Hanson, D. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.

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

[9] Kessler, P. Fast Breakpoints: Design and Implementation. In *Proceedings ACM SIGPLAN'90 Conf. on Programming Languages Design and Implementation*, pages 78–84, 1990.

[10] Sosic, R. and Abramson, D. A. Guard: A Relative Debugger. *Software Practice and Experience*, 27(2):185–206, February 1997.

[11] Wu, L., Mirani, R., Patil H., Olsen, B., and Hwu, W.W. A New Framework for Debugging Globally Optimized Code. In *Proceedings ACM SIGPLAN'99 Conf. on Programming Languages Design and Implementation*, pages 181–191, 1999.