Comparison Checking: An Approach to Avoid Debugging of Optimized Code *

Clara Jaramillo, Rajiv Gupta**, and Mary Lou Soffa

Department of Computer Science, University of Pittsburgh
Pittsburgh, PA 15260, U.S.A.
`cij,gupta,soffa@cs.pitt.edu`

Abstract. We present a novel approach to avoid the debugging of optimized code through comparison checking. In the technique presented, both the unoptimized and optimized versions of an application program are executed, and computed values are compared to ensure the behaviors of the two versions are the same under the given input. If the values are different, the comparison checker displays where in the application program the differences occurred and what optimizations were involved. The user can utilize this information and a conventional debugger to determine if an error is in the unoptimized code. If the error is in the optimized code, the user can turn off those offending optimizations and leave the other optimizations in place. We implemented our comparison checking scheme, which executes the unoptimized and optimized versions of C programs, and ran experiments that demonstrate the approach is effective and practical.

1 Introduction

Although optimizations are important in improving the performance of programs, an application programmer typically compiles a program during the development phase with the optimizer turned off. After the program is tested and apparently free of bugs, it is ready for production. The user then compiles the program with the optimizer turned on to take advantage of the performance improvement offered by optimizations. However, when the application is optimized, its semantic behavior may not be the same as the unoptimized program; we consider the semantic behaviors of an unoptimized program and its optimized program version to be the same if all corresponding statements executed in both programs compute the same values under given inputs. In this situation, the programmer is likely to assume that errors in the optimizer are responsible for the change in semantic behavior and, lacking any more information, turn all the optimizations off.

* Supported in part by NSF grants CCR-940226, CCR-9704350, CCR-9808590 and EIA-9806525, and a grant from Hewlett Packard Labs to the University of Pittsburgh.

** Current address: Department of Computer Science,University of Arizona, Tucson, AZ 85721,U.S.A.

Differences in semantic behaviors between unoptimized and optimized program versions are caused by either (1) the application of an unsafe optimization, (2) an error in the optimizer, or (3) an error in the source program that is exposed by the optimization. For instance, reordered operations under certain conditions can cause overflow or underflow or produce different floating point values. The optimized program may crash (e.g., division by zero) because of code reordering. The application of an optimization may assume that the source code being transformed follows a programming standard (e.g., ANSI standard), and if the code does not, then an error can be introduced by the optimization. The optimizer itself may also contain an error in the implementation of a particular optimization. And lastly, the execution of the optimized program may uncover an error that was not detected in the unoptimized program (e.g., uninitialized variable, stray pointer, array index out of bounds). Thus, for a number of reasons, a program may execute correctly when compiled with the optimizer turned off but fail when the optimizer is turned on.

Determining the cause of errors in an optimized program is hampered by the limitations of current techniques for debugging optimized code. Source level debugging techniques for optimized code have been developed but they either constrain debugging features, limit optimizations, modify the code, or burden the user with understanding how optimizations affect the source level program. For example, to locate an error in the optimized program, the user must step through the execution of the optimized program and examine values of variables to find the statement that computes an incorrect value. Unfortunately, if the user wishes to observe the value of a variable at some program point, the debugger may not be able to report this value because the value has not been computed yet or was overwritten. Techniques to recover the values for reporting purposes work in limited situations and for a limited set of optimizations [13, 10, 25, 19, 7, 12, 14, 4, 18, 5, 6, 23, 3, 24, 8].

In this paper, we present *comparison checking*, a novel approach that avoids debugging of optimized code. The comparison checking scheme is illustrated in Figure 1. The user first develops, tests, and debugs the unoptimized program using a conventional debugger and once the program appears free of bugs, the program is optimized. The comparison checker orchestrates the executions of both the unoptimized and optimized versions of a source program under a number of inputs and compares the semantic behaviors of both program versions for each of the inputs; comparisons are performed on values computed by corresponding executed statements from both program versions. If the semantic behaviors are the same and correct, the optimized program can be run with high confidence. On the other hand, if the semantic behaviors differ, the checker displays the statements responsible for the differences and optimizations applied to these statements. The user can use this information and a conventional debugger to determine if the error is in the unoptimized or optimized code. If the user finds the problem to be in the unoptimized version, modifications are performed and the checking is repeated. If the error is in the optimized code, the user can turn off those offending optimizations in the effected parts of the source

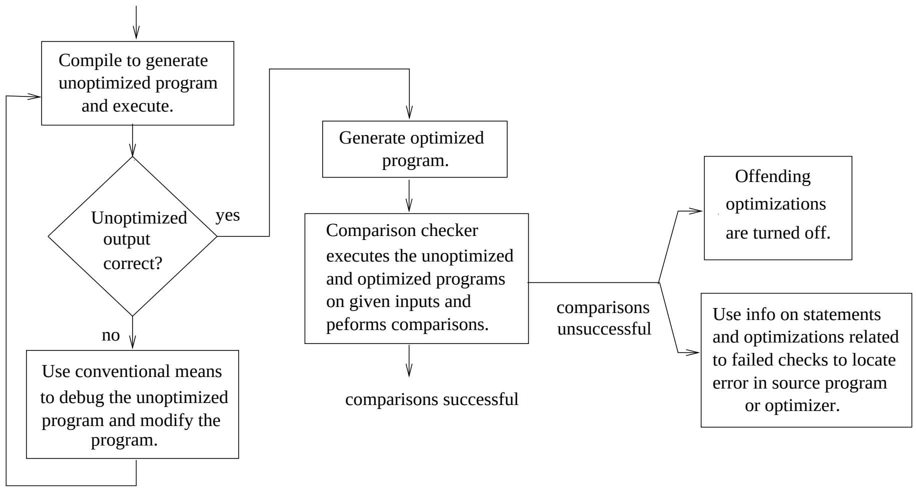

Fig. 1. The Comparison Checking System

program. The difference would be eliminated without sacrificing the benefits of correctly applied optimizations.

Our system can locate the *earliest* point where both programs differ in their semantic behavior. That is, the checker detects the earliest point during execution time when corresponding statement instances should but do not compute the same values. For example, if a difference is due to an uninitialized variable, our scheme detects the first incorrect use of the value of this variable. If a difference is due to the optimizer, this scheme helps locate the statement that was incorrectly optimized. In fact, this scheme proved very useful in debugging our optimizer that we implemented for this work.

The merits of a comparison checking system are as follows.

- The user debugs the unoptimized version of the program, and is therefore not burdened with understanding the optimized code.
- The user has greater confidence in the correctness of the optimized program.
- When a comparison fails, we report the earliest place where the failure occurred and the optimizations that involved the statement. Information about where an optimized program differs from the unoptimized version benefits the user in tracking down the error as well as the optimizer writer in debugging the optimizer.
- A wide range of optimizations including classical optimizations, register allocation, loop transformations, and inlining can be handled by the technique. Many of these optimizations are not supported by current techniques to debug optimized code.
- The optimized code is not modified except for breakpoints, and thus no recompilation is required.

The design of comparison checking has several challenges. We must decide what values computed in both the unoptimized and optimized programs should be compared and how to associate these values in both programs. These tasks are achieved by generating *mappings* between corresponding instances of statements in the unoptimized and optimized programs as optimizations are applied. We must also decide how to execute both programs and when the comparisons should be performed. Since values can be computed out of order, a mechanism must save values that are computed early. These values are saved in a *value pool* and removed when no longer needed. Finally, the above tasks must be automated. The mappings are used to automatically generate *annotations* for the optimized and unoptimized programs. These annotations guide the comparison checker in comparing corresponding values and addresses. Checking is performed for corresponding assignments of values to source level variables and results of corresponding branch predicates. 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 never computed in the optimized code. We implemented the scheme, which executes the unoptimized and optimized versions of C programs. We also present our experience with the system and experimental results demonstrating the practicality of the system.

While our comparison checking scheme is generally applicable to a wide range of optimizations from simple code reordering transformations to loop transformations, this paper focuses on classical statement level optimizations. Optimizations can be performed at the source, intermediate, or target code level, and our checker is language independent. Mappings between the source and intermediate/target code are maintained to report differences in terms of the source level statements.

This paper is organized by demonstrating the usefulness of the comparison checking scheme in Section 2. An overview of the mappings is presented in Section 3. Sections 4 and 5 describe the annotations and the comparison checker. Section 6 provides an example of comparison checking. Section 7 presents experimental results. Related work is discussed in Section 8, and concluding remarks are given in Section 9.

2 Comparison Checking Scenarios

In this section, we demonstrate the usefulness of comparison checking using examples. Consider the unoptimized C program fragment and its optimized version in Figure 2(a). Assume the unoptimized program has been tested, the optimizer is turned on, and when the optimized program executes, it returns incorrect output. The natural inclination of the user is to think that an error in the optimizer caused the optimized program to generate incorrect output. However, using the comparison checker, a difference in the program versions is detected at line 7 in the unoptimized program. The checker indicates that the value 2 is assigned to y in the unoptimized program and the value 22 is assigned to y in the optimized

Unoptimized Program Fragment

```
1)  int a[10];
2)  int b,x,y,i;
     ...
3)  i = 1;
4)  while (i <= 10) {
5)     fscanf(...&a[i]);
6)     x = 2 * i;
7)     y = a[i]+2;
8)     if (x < y)
9)        printf(... x);
10)    else
11)       printf(...2*y);
12)    i = i + 1;
13) }
```

Optimized Program Fragment

```
1)  int a[10];
2)  int x,y,i;
     ...
3)  i = 1;
4)  while (i <= 10) {
5)     fscanf(...&a[i]);
6)     x = 2 * i;
7)     y = a[i]+2;
8)     if (x < y)
9)        printf(... x);
10)    else
11)       printf(...2*y);
12)    i = i + 1;
13) }
```

(a) Example 1

Unoptimized Program Fragment

```
1) void one(x,y,z)
2) int x, y, z;
     ...
3) if (x == y) {
     ...
4)    z = 5;
5) }
6) else {
     ...
7) }
8) x = x * z;
```

Optimized Program Fragment

```
1) void one(x,y,z)
2) int x, y, z;
     ...
3) if (x == y) {
     ...
4)    z = 5;
5) }
6) else {
     ...
7) }
8) x = x * 5;
```

(b) Example 2

Fig. 2. Program examples for comparison checking

program at line 7 during loop iteration 10. The checker also indicates that no optimizations were applied to the statement at line 7. To determine if there is an error in the unoptimized program or if perhaps a prior optimization caused the problem, the user utilizes a conventional debugger and places a breakpoint in the unoptimized program at line 7. Upon the last iteration of the loop, the user examines the value of i, which is 10, and realizes that 10 is not within the bounds of array a (C assigns subscripts $0 \ldots 9$). At this point, the user realizes that the error is in the original program. The user can fix the original program, generate the new optimized version, and repeat the checking. Notice that the user does not need to examine the optimized program.

What caused the optimized program to generate incorrect output? The cause was an optimization that changed the memory layout of the program. Consider the execution time behavior of both program versions. The loop stores values into array a at statement 5 and overruns the upper bound of a when i is 10. In the unoptimized program, the assignment to $a[i]$ when i is 10 actually stores a value in b. Assuming b is not used in the unoptimized program, the output of the program is correct. However, in the optimized program in which storage for b is removed because variable b is dead, the assignment to $a[i]$ when i is 10 actually stores a value in x. Since the same value of x is overwritten at line 6 but needed in subsequent statements, the output of the optimized code is incorrect.

Even when the outputs generated by the unoptimized and optimized programs are both the same and correct, comparing the internal behaviors of the unoptimized and optimized programs can still help users detect errors in the original program. This would happen when an error in the unoptimized program is unmasked in the optimized program and affects the output only on certain inputs. Assume the unoptimized program and optimized program in Figure 2(a) output the same values at statement 9. The error in the program described above does not affect this output. However, using the comparison checker, a difference in the programs is detected at line 7 in the unoptimized program. As mentioned above, the user can utilize the information supplied by the checker and a conventional debugger to determine that at line 7, array a is indexing out of its bounds.

Differences in the behavior of the unoptimized and optimized programs can also be caused by errors in the way optimizations are applied. Consider the example in Figure 2(b). Using the checker and assuming x, y, and z upon entry to the procedure *one* are $1, 2$, and 3, respectively, a difference is detected in the internal behavior of both programs at line 8 in the unoptimized program. The checker indicates that the value 3 is assigned in the unoptimized program and the value 5 is assigned in the optimized program. The checker indicates that constant propagation was applied to this statement in the optimized program and an operand in line 8 was replaced by the constant 5. The user can once again use a conventional debugger to determine if there is an error in the original program. The user places a breakpoint in the unoptimized program at line 8. When the unoptimized program execution reaches the breakpoint, the user examines the values of the operands at line 8 and notices none of the operand's values are 5. The user concludes the value 5 should not have replaced an operand at line 8 and therefore, the constant propagation optimization has been incorrectly applied. The user disables the constant propagation optimization, the checking is repeated, and no differences are reported. The difference was eliminated without sacrificing the benefits of other correctly applied optimizations.

3 Mappings

To compare values computed in both the unoptimized and optimized programs, we need to determine the corresponding statement instances that compute the 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 [16], 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. Since optimizations can be applied in any order and as many times as desired, the mappings also summarize the effects of all previously applied optimizations. In this section, we use an example to describe the mappings used by the comparison checking scheme to support code optimized with classical optimizations.

In Figure 3, the unoptimized program and the optimized program version as well as the mappings are shown. For ease of understanding, the source level statements shown are simple. The mapping technique applies when optimizations are applied at the source, intermediate, or target code level. The mappings are

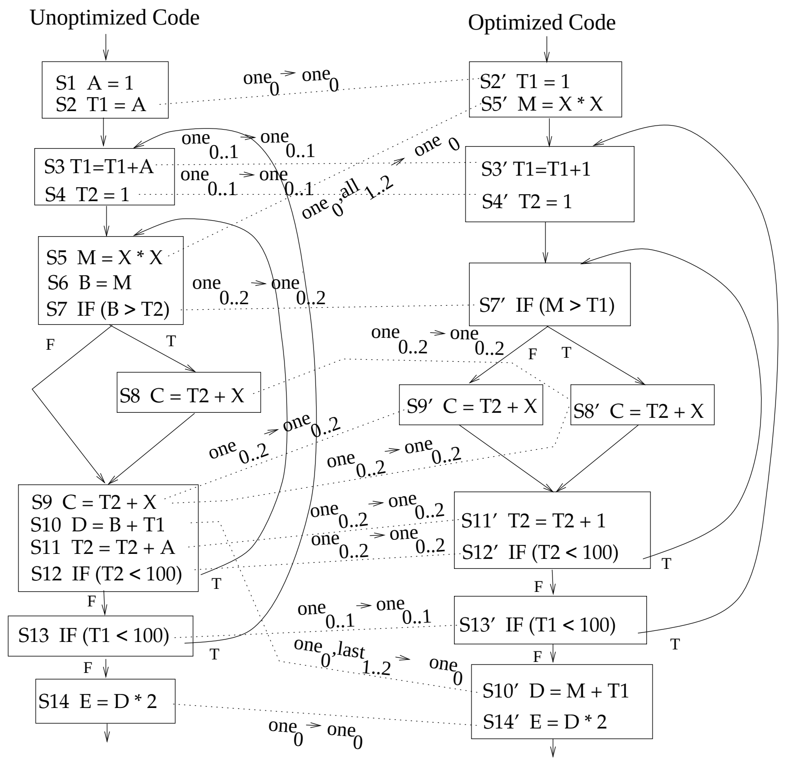

Fig. 3. Mappings for Unoptimized and Optimized Code

illustrated by labeled dotted edges between corresponding statements in both programs. 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.

Mapping labels identify the instances in the unoptimized program and the corresponding instances in the optimized program. If there is an one-to-one correspondence between the instances, then the number of instances is the same and corresponding instances appear in the same order. If the number of instances is not the same, a consecutive subsequence of instances in one sequence corresponds to a single instance in the other.

Since code optimizations move, modify, delete, and add statements in a program, the number of instances of a statement can increase, decrease, or remain the same in the optimized program as compared to the unoptimized program. If a statement is moved out of a loop, then the statement will execute fewer times in the optimized code. For example, in loop invariant code motion (see statements $S5$ and $S5'$), the statement moved out of the loop will execute fewer times and thus **all** the instances of statement $S5$ in the loop in the unoptimized code must map to **one** instance of statement $S5'$ in the optimized code. If a statement is moved below a loop, for example by applying partial dead code elimination (see statements $S10$ and $S10'$), then only the **last** instance of statement $S10$ in the loop in the unoptimized code is mapped to **one** instance of statement $S10'$ in the optimized code. If the optimization does not move the statement across a loop boundary (see statements $S2$ and $S2'$), then the number of instances executed in the unoptimized and optimized code is the same and thus **one** instance of statement $S2$ in the unoptimized code is mapped to **one** instance of statement $S2'$ in the optimized code. The removal of statements (see statements $S1$ and $S6$) can cause mappings to be removed because there is no correspondence between a statement in the unoptimized code to a deleted statement in the optimized code.

4 Annotations

Code annotations guide the comparison checking of values computed by corresponding statement instances from the unoptimized and optimized code. Annotations (1) identify program points where comparison checks should be performed, (2) indicate if values should be saved in a value pool so that they will be available when checks are performed, and (3) indicate when a value currently residing in the value pool can be discarded since all checks involving the value have been performed. The selection and placement of annotations are independent of particular optimizations and depend only on which and how statement instances correspond and the relative positions of corresponding statements in both the unoptimized and optimized programs. Data flow analysis, including reachability and postdominance, is used to determine where and what annotations to use [15]. Annotations are placed after all optimizations are performed and target code has been generated, and therefore, the code to emit the annotations can be integrated as a separate phase within a compiler.

Five different types of annotations are needed to implement our comparison checking strategy. In Figure 4(a), annotations shown in dotted boxes, are given for the example in Figure 3. In the following description, S_{uopt} indicates a statement in the unoptimized code and S_{opt} a statement in the optimized code.

Check S_{uopt} annotation: This annotation is associated with a program point in the optimized code to indicate a check of a value computed by statement S_{uopt} is to be performed. The corresponding value to be compared is the result

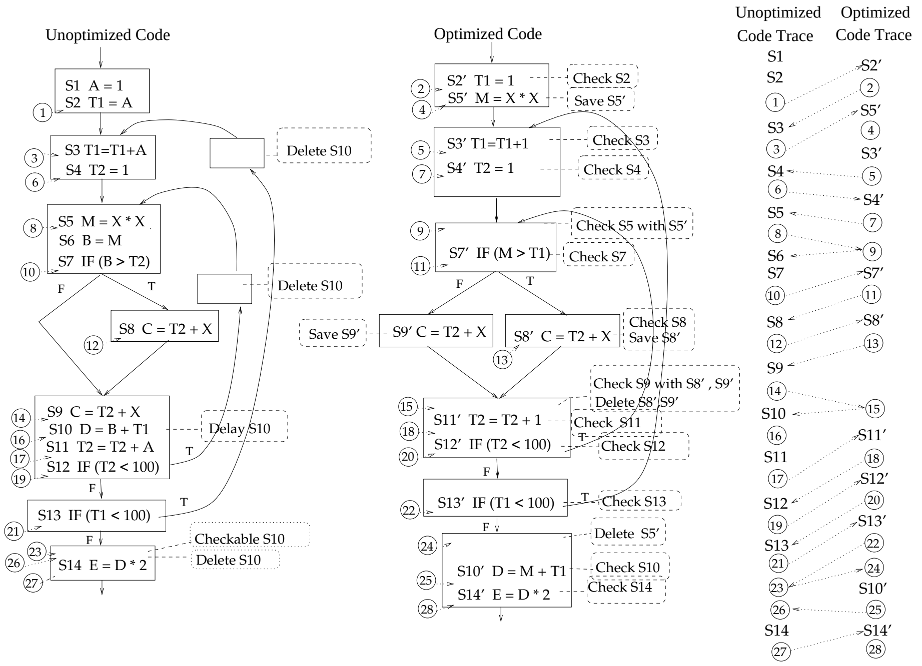

Fig. 4. (a) Annotated Unoptimized and Optimized Code (b) Check Traces

of the most recently executed statement in the optimized code. For example, in Figure 4(a), the annotation *Check* $S2$ is associated with $S2'$.

A variation of this annotation is the **Check** S_{uopt} **with** $S_i, S_j, \ldots$ annotation, which is associated with a program point in the optimized code to indicate a check of a value computed by statement S_{uopt} is to be performed with a value computed by one of S_{uopt}'s corresponding statements $S_i, S_j, \ldots$ in the optimized program. The corresponding value to be compared is either the result of the most recently executed statement in the optimized code or is in the value pool.

Save S_{opt} annotation: If a value computed by a statement S_{opt} cannot be immediately compared with the corresponding value computed by the unoptimized code, then the value computed by S_{opt} must be saved in the value pool. In some situations, a value computed by S_{opt} must be compared with multiple values computed by the unoptimized code. Therefore, it must be saved until all those values have been computed and compared. The annotation *Save* S_{opt} is associated with S_{opt} to ensure the value is saved. In Figure 4(a), the statement $S5$ in the unoptimized code, which is moved out of the loops by invariant code motion, corresponds to statement $S5'$ in the optimized code. The value computed by $S5'$ cannot be immediately compared with the corresponding values computed by $S5$ in the unoptimized code because $S5'$ is executed prior to the execution of $S5$. Thus, the annotation *Save* $S5'$ is associated with $S5'$.

Delay S_{uopt} and **Checkable** S_{uopt} annotations: If the value computed by the execution of a statement S_{uopt} cannot be immediately compared with the corresponding value computed by the optimized code because the correspondence between the values cannot be immediately established, then the value of S_{uopt} must be saved in the value pool. The annotation *Delay* S_{uopt} is associated with S_{uopt} to indicate the checking of the value computed by S_{uopt} should be delayed, saving the value in the value pool. The point in the unoptimized code at which checking can finally be performed is marked using the annotation **Checkable** S_{uopt}.

In some situations, a delay check is needed because the correspondence between statement instances cannot be established unless the execution of the unoptimized code is further advanced. In Figure 4(a), statement $S10$ inside the loops in the unoptimized code is moved after the loops in the optimized code by partial dead code elimination. In this situation, only the value computed by statement $S10$ during the last iteration of the nested loops is to be compared with the value computed by $S10'$. However, an execution of $S10$ corresponding to the last iteration of the nested loops can only be determined when the execution of the unoptimized code exits the loops. Therefore, the checking of $S10$'s value is delayed.

Delete S: This annotation is associated with a program point in the unoptimized/optimized code to indicate a value computed previously by S and stored in the value pool can be discarded. Since a value may be involved in multiple checks, a delete annotation must be introduced at a point where all relevant checks would have been performed. In Figure 4(a), the annotation *Delete* $S5'$ is introduced after the loops in the optimized code because at that point, all values computed by statement $S5$ in the unoptimized code will certainly have been compared with the corresponding value computed by $S5'$ in the optimized code.

Check-self S: This annotation is associated with a program point in the unoptimized/optimized code and indicates that values computed by S must be compared against each other to ensure the values are the same. This annotation causes a value of S to be saved in the value pool.

5 Comparison Checker

The comparison checker compares values computed by both the unoptimized and optimized program executions to ensure the semantic behaviors of both programs are the same. The unoptimized and optimized programs are annotated with actions that guide the comparison checking process. Once a program point is reached, the actions associated with the annotation are executed by the comparison checker. To avoid modifying the unoptimized and optimized programs, breakpoints are used to extract values from the unoptimized and optimized programs as well as activate annotations.

A high level conceptual overview of the comparison checker algorithm is given in Figure 5. Execution begins in the unoptimized code and proceeds un-

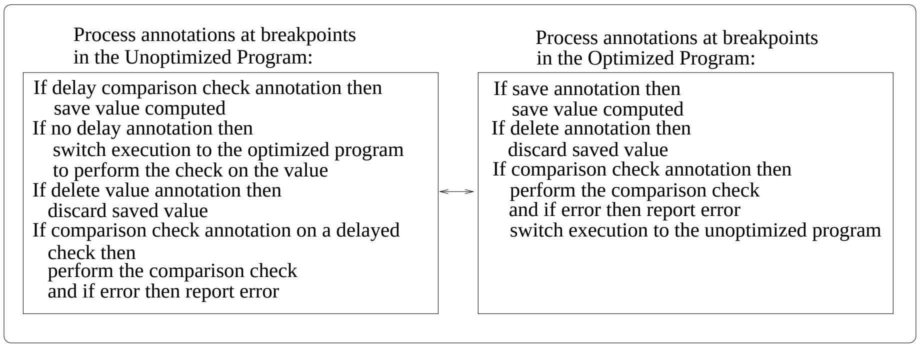

Fig. 5. Comparison Checker Algorithm

til a breakpoint is reached. Using the annotations, the checker can determine if the value computed can be checked at this point. If so, the optimized program executes until the corresponding value is computed (as indicated by an 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 annotations. If annotations indicate 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. Any statement instances eliminated in the optimized code are not checked.

6 Comparison Checking Scheme Example

Consider the unoptimized and optimized program segments in Figure 4. Assume all the statements shown are source level statements and loops execute for a single iteration. Breakpoints are indicated by circles. The switching between the unoptimized and optimized program executions by the checker is illustrated by the traces. The traces include the statements executed as well as the breakpoints (circled) where annotations are processed. The arrows indicate the switching between programs.

The unoptimized program starts to execute with $S1$ and continues executing without checking, as $S1$ was eliminated from the optimized program. After $S2$ executes, breakpoint 1 is reached and the checker determines from the annotation that the value computed 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.

7 Implementation of the Comparison Checking Scheme

We implemented our **C**omparison checking of **OP**timized code scheme, called **COP**, to test our algorithms for instruction mapping, annotation placement, and checking. Lcc [9] was used as the compiler for the application program and was extended to include a set of optimizations, namely loop invariant code motion, dead code elimination, partial redundancy elimination, copy propagation, and constant propagation and folding. On average, the optimized code 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 generate a file containing 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. 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 [21]. Breakpointing is implemented using fast breakpoints [17].

Experiments were performed to assess the practicality of COP. Our main concerns were usefulness as well as cost of the comparison checking scheme. COP was found to be very useful in actually debugging our optimizer. 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

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.

Table 1. Execution Times (minutes:seconds)

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, we were interested in the slow downs of the unoptimized and optimized programs and the speed of the comparison checker. COP performs *on-the-fly* checking during the execution of both programs. Both value and address comparisons are performed. In our experiments, we ran COP 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. We ran some of the integer Spec95 benchmarks as well as some smaller test programs.

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. We found that the performance of the checker is bounded by the processing platform and speed of the network. A faster processor and 100 Mb network would considerably lower these times. In fact, we ran COP on a 333 MHz Pentium Pro processor and found the performance to be on average 6 times faster in terms of CPU time. We did not have access to a faster network. We measured the pool size during our experiments and found it to be fairly small. If addresses are not compared, the pool size contains less than 40 values for all programs. If addresses are compared, then the pool size contains less than 1900 values.

8 Related Work

The problem of debugging optimized code has long been recognized [13, 20], with most of the previous work focusing on the development of source level debuggers of optimized code [13, 10, 25, 19, 7, 12, 14, 4, 4, 18, 5, 6, 23, 3] that use static analysis techniques to determine whether expected values of source level variables are reportable at breakpoints. Mappings that track an unoptimized program to its optimized program version are statically analyzed to determine the proper placement of breakpoints as well as to determine if values of source level variables are reportable at given breakpoints. However, all values of source level variables are not reportable and optimizations are restricted. One reason is that the mappings are too coarse in that statements (and not instances) from the unoptimized program are mapped to corresponding statements in the optimized program. Another reason is that the effectiveness of static analysis in reporting expected values is limited. Recent work on source level debuggers of optimized code utilizes some dynamic information to provide more of the expected behavior of the unoptimized program. By executing the optimized code in an order that mimics the execution of the unoptimized program, some values of variables that are otherwise not reportable by other debuggers can be reported in [24]. However, altering the execution of the optimized program masks certain user and optimizer errors. The technique proposed in [8] can also report some values of variables that are not reportable by other debuggers by timestamping basic blocks to obtain a partial history of the execution path, which is used to precisely determine what variables are reportable at breakpoints. In all of the prior approaches, limitations have been placed on the debugging system by either restricting the type or placement of optimizations, modifying the optimized program, or inhibiting debugging capabilities. These techniques have not found their way into production type environments, and debugging of optimized code still remains a problem.

In [16], we developed a mapping technique for associating corresponding statement instances after optimization and describe how to generate mappings that support code optimized with classical optimizations as well as loop transformations. Since we track corresponding statement instances and use dynamic information, our checker can automatically detect and report the earliest source statement that computes a different value in the optimized code as well as report the optimizations applied to that statement.

For understanding optimized code, [22] creates a modified version of the source program along with annotations to display the effects of optimizations. Their annotations are higher level than ours as their focus is on program understanding rather than comparison checking.

The goal of our system is not to build a debugger for optimized code but to verify that given an input, the semantic behaviors of both the unoptimized and optimized programs are the same. The most closely related work to our approach is Guard [21, 2, 1], which is a relative debugger, but not designed to debug optimized programs. Using Guard, users can compare the execution of one program, the reference program, with the execution of another program, the development version. Guard requires the user to formulate assertions about the key data structures in both versions which specify the locations at which the data structures should be identical. The relative debugger is then responsible for managing the execution of the two programs and reporting any differences in values. The technique does not require any modifications to user programs and can perform comparisons on-the-fly. The important difference between Guard and COP is that in Guard, the user essentially has to manually insert all of the mappings and annotations, while this is done automatically in COP. Thus using COP, the optimized program is transparent to the user. We also are able to check the entire program which would be difficult in Guard since that would require the user inserting all mappings. In COP, we can easily restrict checking to certain regions or statements as Guard does. We can also report the particular optimizations involved in producing erroneous behavior.

The Bisection debugging model [11] was also designed to identify semantic differences between two versions of the same program, one of which is assumed to be correct. The bisection debugger attempts to identify the earliest point where the two versions diverge. However, to handle the debugging of optimized code, the expected values of source level variables must be reportable at all breakpoints. This is not an issue in COP because expected values of source level variables are available at comparison points, having been saved before they are overwritten.

9 Conclusion

Optimizations play an important role in the production of quality software. Each level of optimization can improve program performance by approximately 25%. Moreover, optimizations are often required because of the time and memory constraints imposed on some systems. Thus, optimized code plays, and will continue to play, an important role in the development of high performance systems.

Our work adds a valuable tool in the toolkit of software developers. This tool verifies that given an input, the semantic behaviors of both the unoptimized and optimized program versions are the same. When the behaviors differ, information to help the programmer locate the cause of the differences is provided. Even when the outputs generated by the unoptimized and optimized programs are correct, comparing the internal behaviors of the unoptimized and optimized programs can help programmers detect errors in the original program. We implemented this tool, which executes the unoptimized and optimized versions of C programs, and ran experiments that demonstrate the approach is effective and practical.

Our checking strategy can also be used to check part of the execution of an application program. For example, during testing, programs are typically executed under different inputs, and checking the entire program under every input may be redundant and unnecessary. Our tool can check regions of the application program, as specified by the user. In this way, the technique can scale to larger applications. Furthermore, the mapping and annotation techniques can be used in different software engineering applications. For example, optimized code is difficult to inspect in isolation of the original program. The mappings enable the inspection of optimized code guided by the inspection of the unoptimized code. Also, the same types of annotations can be used for implementing a source level debugger that executes optimized code and saves runtime information so that all values of source level variables are reportable. We are currently developing such a debugger.

References

1. Abramson, D. and Sosic, R. A Debugging Tool for Software Evolution. *CASE-95, 7th International Workshop on Computer-Aided Software Engineering*, pages 206–214, July 1995.
2. Abramson, D. and Sosic, R. A Debugging and Testing Tool for Supporting Software Evolution. *Journal of Automated Software Engineering*, 3:369–390, 1996.
3. Adl-Tabatabai, A. and Gross, T. Source-Level Debugging of Scalar Optimized Code. In *Proceedings ACM SIGPLAN'96 Conf. on Programming Languages Design and Implementation*, pages 33–43, May 1996.
4. Brooks, G., Hansen, G.J., and Simmons, S. A New Approach to Debugging Optimized Code. In *Proceedings ACM SIGPLAN'92 Conf. on Programming Languages Design and Implementation*, pages 1–11, June 1992.
5. Copperman, M. Debugging Optimized Code: Currency Determination with Data Flow. In *Proceedings Supercomputer Debugging Workshop '92*, October 1992.
6. Copperman, M. Debugging Optimized Code Without Being Misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, 1994.
7. Coutant, D.S., Meloy, S., and Ruscetta, M. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings ACM SIGPLAN'88 Conf. on Programming Languages Design and Implementation*, pages 125–134, June 1988.
8. 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.
9. Fraser, C. and Hanson, D. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.
10. Fritzson, P. A Systematic Approach to Advanced Debugging through Incremental Compilation. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 130–139, 1983.
11. Gross, T. Bisection Debugging. In *Proceedings of the AADEBUG'97 Workshop*, pages 185–191, May, 1997.
12. Gupta, R. Debugging Code Reorganized by a Trace Scheduling Compiler. *Structured Programming*, 11(3):141–150, 1990.
13. Hennessy, J. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.
14. Holzle, U., Chambers, C., and Ungar, D. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings ACM SIGPLAN'92 Conf. on Programming Languages Design and Implementation*, pages 32–43, June 1992.

15. Jaramillo, C. *Debugging of Optimized Code Through Comparison Checking.* PhD dissertation, University of Pittburgh, 1999.
16. 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.
17. Kessler, P. Fast Breakpoints: Design and Implementation. In *Proceedings ACM SIGPLAN'90 Conf. on Programming Languages Design and Implementation*, pages 78–84, 1990.
18. Pineo, P.P. and Soffa, M.L. Debugging Parallelized Code using Code Liberation Techniques. *Proceedings of ACM/ONR SIGPLAN Workshop on Parallel and Distributed Debugging*, 26(4):103–114, May 1991.
19. Pollock, L.L. and Soffa, M.L. High-Level Debugging with the Aid of an Incremental Optimizer. In *21st Annual Hawaii International Conference on System Sciences*, volume 2, pages 524–531, January 1988.
20. Seidner, R. and Tindall, N. Interactive Debug Requirements. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 9–22, 1983.
21. Sosic, R. and Abramson, D. A. Guard: A Relative Debugger. *Software Practice and Experience*, 27(2):185–206, February 1997.
22. Tice, C. and Graham, S.L. OPTVIEW: A New Approach for Examining Optimized Code. *Proceedings of ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, June 1998.
23. Wismueller, R. Debugging of Globally Optimized Programs Using Data Flow Analysis. In *Proceedings ACM SIGPLAN'94 Conf. on Programming Languages Design and Implementation*, pages 278–289, June 1994.
24. 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.
25. Zellweger, P.T. An Interactive High-Level Debugger for Control-Flow Optimized Programs. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 159–171, 1983.