
Validation of GCC optimizers through trace generation



Aditya Kanade*, Amitabha Sanyal, Uday P. Khedker

*Dept. of Computer Science and Engineering, IIT Bombay.
Email: kanade@seas.upenn.edu {as,uday}@cse.iitb.ac.in*

SUMMARY

The translation validation approach involves establishing semantics preservation of individual compilations. In this paper, we present a novel framework for translation validation of optimizers. We identify a comprehensive set of primitive program transformations that are commonly used in many optimizations. For each primitive, we define soundness conditions which guarantee that the transformation is semantics preserving. This framework of transformations and soundness conditions is independent of any particular compiler implementation and is formalized in PVS.

An optimizer is instrumented to generate the trace of an optimization run in terms of the predefined transformation primitives. The validation succeeds if (1) the trace conforms to the optimization and (2) the soundness conditions of the individual transformations in the trace are satisfied. The first step eliminates the need to trust the instrumentation. The soundness conditions are defined in a temporal logic and therefore the second step involves model checking. Thus the scheme is completely automatable.

We have applied this approach to several intraprocedural optimizations of RTL intermediate code in GCC v4.1.0, namely, loop invariant code motion, partial redundancy elimination, lazy code motion, code hoisting, and copy and constant propagation for sample programs written in a subset of the C language. The validation does not require information about program analyses performed by GCC. Therefore even though the GCC code base is quite large and complex, instrumentation could be achieved easily. The framework requires an estimated 21 lines of instrumentation code and 140 lines of PVS specifications for every 1000 lines of the GCC code considered for validation.

KEY WORDS: Compiler optimization, Translation validation, GCC, Temporal logic, PVS

1. Introduction

A compiler optimizer analyzes and transforms programs to improve their run-time behavior. This allows programmers to focus on functionality of programs without having to bother

*Present address: Department of Computer and Information Science, University of Pennsylvania.

about efficiency of the generated code. Optimizers have therefore become an integral part of modern compilers. However, a mistake in the design or the implementation of an optimizer can proliferate in the form of bugs in the softwares compiled through it.

The issue of soundness of optimizers is usually addressed at two levels: (1) One time guarantees are obtained at the design level by verifying optimization specifications and (2) runtime guarantees are obtained at the implementation level by validating optimization runs.

Both these approaches involve proofs of semantic equivalence between the input and the optimized programs. However, they are usually tedious. Even in the case of validation where semantic equivalence is to be shown for a particular execution, it cannot be accomplished with ease. This complexity can be conquered by taking advantage of the fact that optimizations with similar objectives employ similar program transformations. For example, “replacement of some occurrences of an expression by a variable” is a transformation which is common to optimizations like common subexpression elimination, lazy code motion, loop invariant code motion, and several others whose aim is to avoid unnecessary recomputations of a value.

This observation led to identification of transformation primitives and soundness conditions, and their use in verification of optimization specifications [15, 14, 13]. A *transformation primitive* denotes a small-step program transformation that is used in many optimizing transformations. These primitives can thus be used to specify a large class of optimizations by sequential composition. The *soundness condition* for a transformation primitive is a condition on programs input to the primitive which if satisfied implies that the transformed program is semantically equivalent to the input program. The soundness conditions essentially capture the context dependent patterns in proofs of semantics preservation for the transformations. Proving sufficiency of soundness conditions for semantics preservation under the respective transformations is a one time affair and is independent of any optimization. Since the primitives are small-step transformations, these proofs are much easier than similar proofs for optimizations. This approach reduces proving soundness of an optimization to merely showing that soundness conditions of the underlying primitives are satisfied on the versions of the input program on which they are applied. This is much simpler than directly proving semantics preservation for each optimization.

In our opinion, this compositional view simplifies the design, implementation, and soundness proofs at both specification (verification) and implementation (validation) levels. In fact, the GCC (v4.1.0) implementation is a witness to the merit of this view. GCC optimizes a program by applying a sequence of smaller transformation routines to it. Using the consistency in our view and the GCC implementation, we have developed a novel validation scheme for GCC optimizers. We instrument GCC to generate traces that describe optimizations as sequences of predefined transformations primitives. We then validate an optimization by checking (1) whether the generated trace conforms to the optimization performed and (2) whether the soundness conditions of the individual transformations in the trace are satisfied. The first step eliminates the need to trust the instrumentation and the second step avoids the need to derive a proof of semantic equivalence between unoptimized and optimized programs.

Given the size and complexity of the GCC code, the task of instrumenting GCC optimizers appears to be daunting. Typically, program analyses and in particular, profitability heuristics, are the most complex and largest parts of optimizer implementations whereas optimizing transformations constitute only a fraction of the actual code. Since semantics preservation of

a transformation is established by checking its soundness condition, our approach does not require any information about program analyses. Consequently, the task of instrumenting the compiler involves examining and instrumenting only the optimizing transformation routines and is therefore easy. For instance, various global common subexpression elimination algorithms are implemented in the GCC source file `gcse.c`. It consists of around 6800 lines of C code whereas the optimizing transformation routines `hoist_code`, `pre_gcse`, and `cprop` consist of only 150, 50, and 15 lines of code respectively. The rest of the code is concerned with data structure implementations, analysis, and book-keeping operations. Our approach is therefore more practical and lightweight than approaches which require an instrumentation of a compiler to generate annotations for the target code [29] or to generate proofs of correctness [24].

The framework of transformation primitives and their soundness conditions is developed in PVS [23] and is independent of any particular optimizer implementation. A trace generated by an instrumented GCC optimizer is converted into a PVS theory and interpreted using the conceptual framework of transformation primitives and soundness conditions. The soundness conditions are expressed in a temporal logic, called Computational Tree Logic with branching past (CTL_{bp}) [16]. The PVS ground evaluator is used for evaluating program transformations and model checking the soundness conditions.

In this paper, we highlight the practical issues encountered while developing the validation framework for GCC and the approaches we used to address them. The present implementation is aimed at estimating the cost of instrumenting a real compiler like GCC and usability of the framework in terms of coverage of various optimizer implementations using only a small set of transformation primitives. The input programs to the compiler are restricted to a small subset of the C language. Scaling up to realistic input programs would require a more comprehensive treatment of the RTL intermediate representation of GCC and a more efficient implementation of the model checking algorithm. Addressing scalability is a future work. Note that the soundness conditions are expressed in CTL_{bp} and the complexity of model checking CTL_{bp} formulae is linear in both the size of the model (program) and the length of the formula [16]. The soundness conditions are specific CTL_{bp} formulae and are small.

The estimate of the cost and usability of the framework is encouraging. The estimated GCC code base that is covered by the validation efforts is around 11900 lines of the source code (including comments). The validation framework requires an estimated 21 lines of instrumentation code and 140 lines of PVS specifications for every 1000 lines of the relevant GCC code base. These specifications also involve some generic background theories (boolean matrix operations) that are not supported by the PVS prelude. Counting instead only the specifications specific to our framework (transformation primitives and soundness conditions), the framework requires around 100 lines of specifications per 1000 lines of GCC code. The PVS specifications are independent of the GCC implementation and can therefore be used with other compiler infrastructures as well, thus mitigating the development cost further. The number of transformation primitives required is also small. The traces generated by 4 optimizer routines are considered. These are expressible as compositions of only 7 transformation primitives.

Contributions. The main contributions of this work are as follows:

1. We present a simple and practical framework for validation of several intraprocedural optimizers implemented in GCC.

2. We have validated optimizations of test programs written in a subset of the C language for all optimization levels of GCC, namely, `01`, `02`, `03`, and `0s`. In particular, we have validated the following bit-vector analysis based optimization routines:
 - loop invariant code motion (`loop.c/move_movables`),
 - partial redundancy elimination (PRE) or global common subexpression elimination (GCSE) through lazy code motion (`gcse.c/pre_gcse`),
 - PRE/GCSE through code hoisting (`gcse.c/hoist_code`), and
 - copy and constant propagation (`gcse.c/cprop`).
3. While experimenting with this framework we have also gained some interesting insights into the functioning of GCC without having to read through the complex code. In fact, we observe that GCC performs some optimizing transformations in a roundabout manner requiring 2–3 steps whereas it can be done in a simpler single step.

Organization. In Section 2, we explain our approach with an example of redundancy elimination performed by GCC. In Section 3, we give an overview of the validation scheme and explain the design of the validation framework for GCC. In Section 4, we define primitive program transformations and characterize semantics preservation in the form of soundness conditions. In Section 5, we explain conversion of RTL intermediate representation to a representation suitable for validation. We also discuss the identification and instrumentation of optimizing transformation routines in the GCC code.

The generated traces are not always in a form that can be validated directly. In Section 6, we discuss some heuristics to generate equivalent traces that can be validated. As we check for conformance of a trace with actual optimization, our scheme is sound even when heuristics are used. To perform validation in the PVS based framework, we convert the traces to PVS theories. In Section 7, we evaluate the cost of development of the framework in terms of the code and specification sizes, discuss the complexity of the validation approach, and present some performance measurements in terms of coverage of the optimizer routines and run-time. In Section 8, we survey related approaches and in Section 9, we summarize the work.

2. An example GCC optimization and its validation

Consider the following C code:

```

if (j <= 0) p = a/b; else q = a/b;
if (m <= i) q = a/b;
return (p+q);

```

Figure 1(a) shows a representation of the RTL intermediate code generated by GCC for the program. The numbers before ‘:’ are statement numbers. The prefix “@” is used to distinguish temporary variables generated by the compiler from numbers and variables in the input program. ‘?’ is a comparison operator and `result` is a special variable containing the return value of a procedure. Figure 1(b) shows the control flow graph (CFG) of the program.

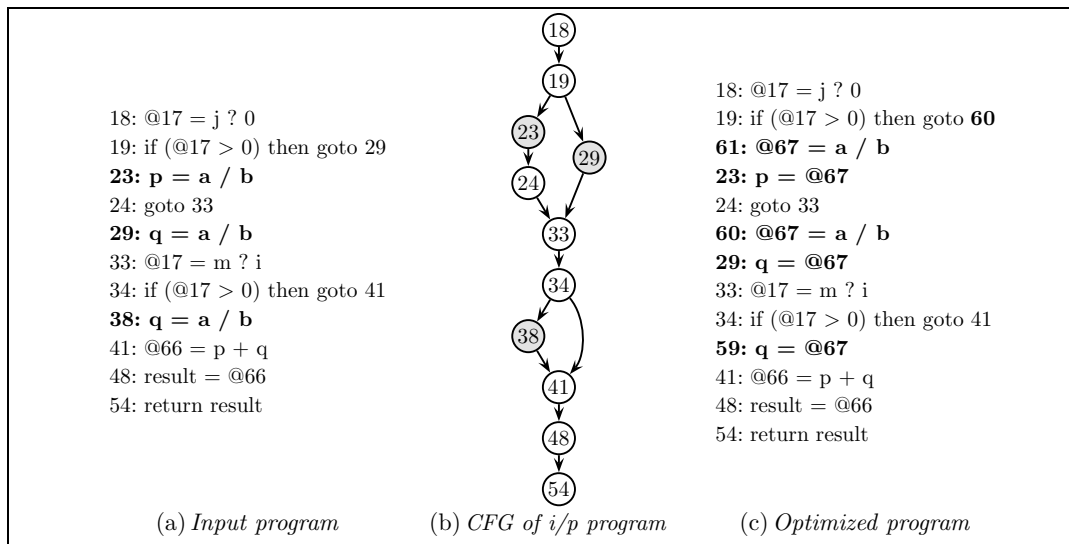


Figure 1. Redundancy elimination performed by GCC v4.1.0

Figure 1(c) shows the program generated by GCC after performing redundancy elimination on the input program (a).

The computation of a/b at program point 38 is redundant since it is computed along all incoming paths i.e. at program points 23 and 29 and its operands (variables a and b) are not assigned in between. This is an example of availability analysis which is used in common subexpression elimination optimization [3].

Conformance of trace. Figure 2 shows the trace of the program transformations applied by GCC while optimizing the program in Figure 1(a) to the program in Figure 1(c). Starting with the input program, each transformation in the sequence transforms the current version of the input program into a new program to which the next transformation is applied.

Transformation T1 is applied to the input program. It inserts a new predecessor program point 61 to program point 23. IP is the transformation primitive for *insertion of predecessors* to a given set of program points. Program point 61 contains SKIP statement. Next, transformation T2 replaces the SKIP statement at program point 61 by assignment $@67 = a/b$. IA is the transformation primitive for *insertion of assignment statements* at a given set of program points. Transformation T3 replaces expression a/b at program point 23 by variable $@67$. RE is the transformation primitive for *replacement of expression* occurrences at a set of program points by a variable. Transformations T4–T6 are similar to T1–T3.

Transformation T7 inserts a successor program point 59 to program point 38. The statement at 59 is SKIP. IS is the transformation primitive for *insertion of successors* to a given set of program points. Transformation T8 inserts assignment $q = @67$ at 59. Finally, transformation T9 deletes program point 38. DS is the transformation primitive for *deletion of statements*.

```

T1 : IP 23 61
T2 : IA 61 ( ASSIGN @67 ( div a b ) )
T3 : RE 23 ( div a b ) @67
T4 : IP 29 60
T5 : IA 60 ( ASSIGN @67 ( div a b ) )
T6 : RE 29 ( div a b ) @67
T7 : IS 38 59
T8 : IA 59 ( ASSIGN q @67 )
T9 : DS 38

```

Figure 2. Trace of program transformations performed by GCC

The program obtained by simulating the trace on the input program in Figure 1(a) is the same as the optimized program in Figure 1(c). Thus the trace conforms to the optimization performed. Note that the trace is generated by the instrumented code whereas the input and the optimized programs are simply recorded during the compilation.

Note: The example programs and trace used here are modified versions of the exact ones generated by the instrumented GCC. In Section 6.1, we discuss the heuristics that we use to convert the exact trace to the one in Figure 2. These heuristics are required because the exact trace cannot be validated directly. However since a (modified) trace is checked for conformance with the actual optimization, the heuristics do *not* compromise the soundness of our scheme.

Soundness of trace. Transformation T1 inserts new predecessor program points which contain SKIP statements. It does not add or remove any paths in the program. It only extends the existing paths. Clearly T1 preserves semantics of the input program. T1 is an application of primitive IP. The soundness condition for IP is true i.e. any application of IP preserves semantics. The soundness conditions of transformation primitives are discussed in Section 4.3.

Transformation T2 inserts assignment $@67 = a/b$ at program point 61. The statement at 61 is a SKIP statement. Further, $@67$ is not used anywhere in the program. Hence an assignment to it does not affect any reaching definitions. Expression a/b is computed at program point 23 which is the only successor to program point 61 (due to T1). Thus the value of a/b at 61 is same as the value of a/b at 23. Therefore the insertion of a computation of a/b at 61 does not compute a new value along any path and transformation T2 preserves semantics.

Transformation T3 replaces the computation of a/b at 23 by $@67$. In the input program to T3, 61 is the only predecessor of 23 (due to T1). 61 contains assignment $@67 = a/b$ (due to T2). Clearly $@67$ has the same value as a/b just before 23. Thus T3 also preserves semantics. Soundness of transformations T4–T6 can be argued in a similar manner.

Transformation T7 inserts new successor program points which contain SKIP statements. Similar to T1, it is easy to see that T7 preserves semantics.

Transformation T8 inserts assignment $q = @67$ at program point 59. In the input program to T8, 38 is the only predecessor of 59 (due to T7). At 38, q is assigned a/b . Along all backward paths starting with predecessors of 38, $@67$ is assigned a/b without any assignment to a , b , and $@67$ in between. Thus the value of a/b is same as the value of $@67$ at 38 and 59. Clearly

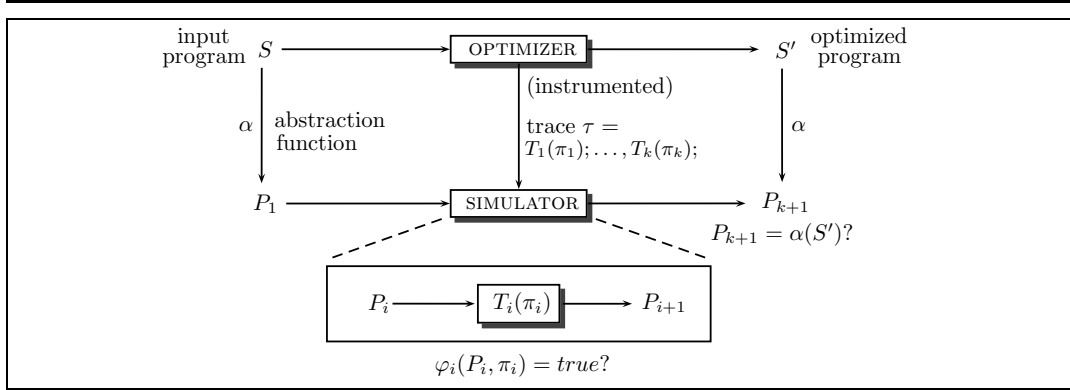


Figure 3. Validation against trace

the assignment to q at 59 does not modify the original value of q that would reach program point 41 where q is used. Hence T8 preserves semantics.

Transformation T9 deletes 38. Since the statement at 38 is an assignment statement, the deletion does not delete any paths in the program. It only deletes program point 38 from the existing paths. The assignment to q at 38 is dead because q is also assigned to at the successor of 38 i.e. program point 59 (due to T8). Thus T9 preserves semantics.

Validation of GCC optimization. Since the trace conforms to the optimization performed by GCC and each of its transformations is semantics preserving, the semantics of the input program is preserved by the GCC optimization.

3. Overview of the validation scheme

In our validation scheme, an optimizer is instrumented to generate a trace τ of its execution as a sequence of appropriately instantiated primitives T_1, \dots, T_k as shown in Figure 3. The program points to which these primitives are applied are π_1, \dots, π_k . P_1 is a control flow graph based abstract representation of the input program S . A transformation $T_i(\pi_i)$ is applied to (abstract) program P_i and results in program P_{i+1} . The optimized program S' is semantically equivalent to the input program S if the following conditions hold:

- (1) The abstract representation of S' matches the output program P_{k+1} obtained by simulating the trace on the input program P_1 i.e. $P_{k+1} = \alpha(S')$.
- (2) For each transformation primitive T_i in the trace τ , the soundness condition φ_i of T_i is satisfied on program P_i i.e. $\varphi_i(P_i, \pi_i) = \text{true}$.

The first step eliminates the need to trust the instrumentation by checking commutativity of S, S', P_1, P_{k+1} mappings. The second step avoids the need to derive proofs of semantic equivalence directly and hence is amenable to automation. We call this scheme *validation against trace*.

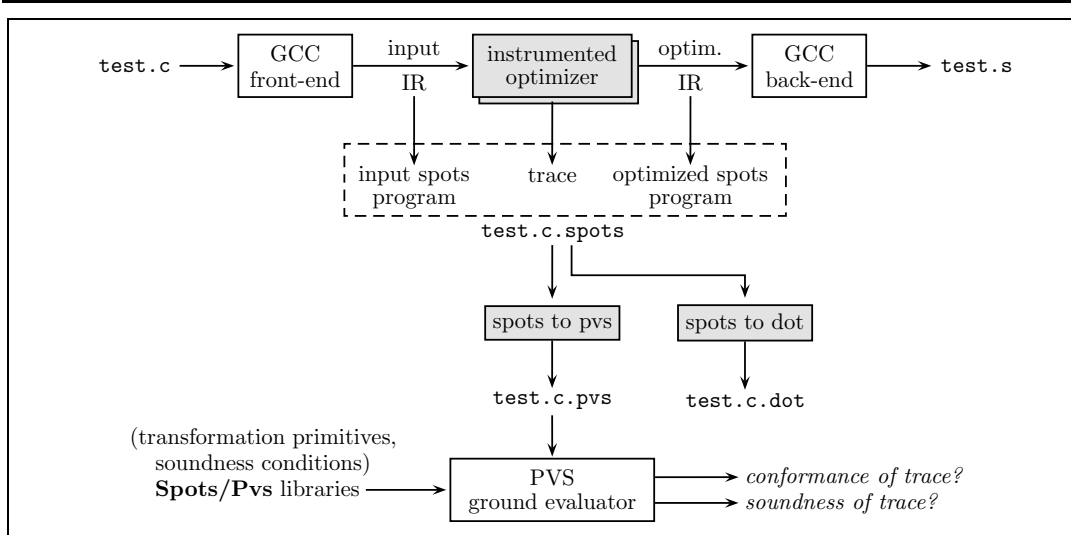


Figure 4. SPOTS/GCC validation framework

Automating this scheme requires a trusted framework for simulating transformation primitives and checking their soundness conditions. In Section 4, we discuss the design of a PVS based framework called SPOTS/PVS for this. SPOTS is an acronym for “System for Proving Optimizing Transformations Sound”. It is used for specification and verification of optimizations [15, 13]. We have developed novel boolean matrix algebraic formulations of transformation primitives and their soundness conditions [14]. These can be directly evaluated in the PVS ground evaluator. This forms the SIMULATOR block in Figure 3. The abstraction function α maps a program (in the intermediate representation of the compiler) to a Kripke structure whose graph is same as the control flow graph of the program and the states are labeled according to the valuations of the local data flow properties. In the SPOTS/GCC framework (Figure 4), it is a syntax-directed translation from the RTL intermediate representation of GCC to the PVS representation. A Kripke structure is then generated in PVS by evaluating the local data flow properties on the program representation.

Figure 4 shows the schematic of the validation framework for GCC, called SPOTS/GCC. We instrument several optimizers of GCC to generate traces of their executions in terms of the transformation primitives defined in the SPOTS/PVS framework. SPOTS/PVS is a compiler independent framework whereas SPOTS/GCC is a validation framework for GCC. The input and the optimized intermediate programs of GCC are converted to PVS theories and subsequently validated in PVS using the SPOTS/PVS libraries.

We consider optimizations of programs in RTL intermediate representation (IR) [2, 1]. We first convert an RTL program to a simplified (spots) format. The spots representation of the input and the optimized programs together with the generated trace (`test.c.spots`) is then converted to a PVS theory (`test.c.pvs`). We also generate verification conditions for the trace. The verification conditions are checked using the PVS ground evaluator. We also generate dot

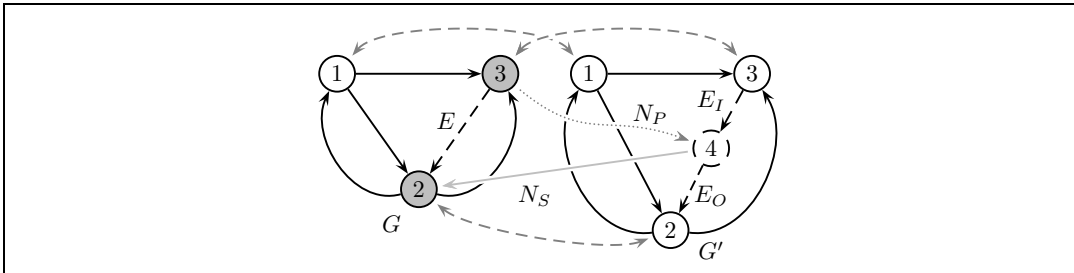


Figure 5. An example of the node addition transformation

representation of all the programs generated by the trace (`test.c.dot`) and then a PS file (`test.c.ps`) for visualizing the actual transformations.

4. The framework of program transformations

A primitive program transformation (or simply a transformation primitive) is defined in terms of: (1) A transformation of the control flow graph and (2) a function to map the statements of the input program to the statements of the transformed program.

4.1. Primitive graph transformations

We define the following primitive transformations of (control flow) graphs: node splitting, node merging, node addition, node deletion, edge addition, edge deletion, and isomorphic transformation [13]. These transformations are defined using boolean matrix algebra and hence are succinct and evaluable. We explain the node addition transformation here.

Node addition transformation. A node addition transformation adds a new node along each edge in a given set of edges E of a graph G . It splits the edges in E and adds the new nodes as successors to the source nodes of the edges in E and as predecessors to the target nodes of the edges in E . The rest of the edges of G are preserved. For example, consider the two graphs shown in Figure 5. G' is obtained by adding node 4 along edge $\langle 3, 2 \rangle$ of G . Edge $\langle 3, 2 \rangle$ is split into two edges $\langle 3, 4 \rangle$ and $\langle 4, 2 \rangle$ making node 4 a successor of node 3 and a predecessor of node 2. The correspondence relation C shown by the dashed gray arrows from right to left denotes the correspondence between the nodes of the transformed graph and the input graph. The lightgray solid arrows from right to left marked as N_S map the newly added nodes to the target nodes of the edges in E . The lightgray dotted arrows from left to right marked as N_P map the source nodes of the edges in E to the newly added nodes.

The correspondence between the edges of the two graphs can be traced diagrammatically. Edge $\langle 3, 4 \rangle$ is obtained by following the C arrow from node 3 of G' to node 3 of G and then following the N_P arrow from node 3 to node 4 of G' . Edge $\langle 4, 2 \rangle$ is obtained by following the N_S arrow from node 4 of G' to node 2 of G and then following the \widehat{C} arrow from node 2 of

G to node 2 of G' . In order to form edges corresponding to the edges of G , we traverse all the edges of G except the edges belonging to E . The composition of edges and arrows can be expressed by boolean matrix multiplication.

Let us denote a graph G by a pair (N, A) where N is the set of nodes and A is the adjacency matrix representation of the edges of G . In the following definition, matrix multiplication is denoted by ' \cdot ', matrix transpose by ' $\widehat{\cdot}$ ', and matrix addition by ' $+$ '.

Definition 1 (Node Addition) *The transformation of a graph $G = (N, A)$ to a graph $G' = (N', A')$ is called a node addition transformation if*

1. *The correspondence relation $C \subseteq N' \times N$ (denoted as a $|N'| \times |N|$ boolean matrix) is a partial, onto, and one-to-one relation and*
2. *There exist a set E of edges of G denoted as a $|N| \times |N|$ matrix, a $|N| \times |N'|$ matrix N_P , and a $|N'| \times |N|$ matrix N_S such that the following conditions hold:*

- (a) $E \leq A$,
- (b) N_P is a total and onto relation from the source nodes of E to the new nodes in G' ,
- (c) N_S is a total and onto relation from the new nodes in G' to the target nodes of E ,
- (d) $E = N_P \cdot N_S$, and
- (e) $(C \cdot A \cdot \widehat{C} - C \cdot E \cdot \widehat{C}) + \underbrace{C \cdot N_P}_{E_I} + \underbrace{N_S \cdot \widehat{C}}_{E_O} = A'$.

The correspondences N_P and N_S and the edges in E_I and E_O are shown in Figure 5. The relation C is partial since it does not relate the new nodes with any nodes of the input graph.

4.2. Primitive program transformations

We define the following primitive program transformations where the transformations of the control flow graph are defined in terms of the primitive graph transformations:

1. An *insertion of predecessors* (IP) transformation inserts a new predecessor program point each to a given set of program points.
2. An *insertion of successors* (IS) transformation inserts a new successor program point each to a given set of program points.
3. An *edge splitting* (SE) transformation splits a set of edges and inserts a new program point along them.

The transformation of the control flow graph for IP, IS, and SE primitives is defined as a special case of the node addition transformation. The statement at the newly inserted program points is `skip`. For IP and SE transformations, each jump statement in the input program whose target is one of the program points in the given set is changed so that the target is the corresponding newly inserted program point in the transformed program. The statements at other program points are not changed. We restrict the applications of these primitives in order to preserve the consistency between the control flow and contents of the program. For example, we do allow insertion of a successor to a conditional statement.

We now explain the insertion of predecessors transformation with an example. Consider the two programs shown in Figure 6 such that $\text{prog}' = \text{IP}(\text{prog}, \text{succs}, \text{newpoints})$. The program prog'

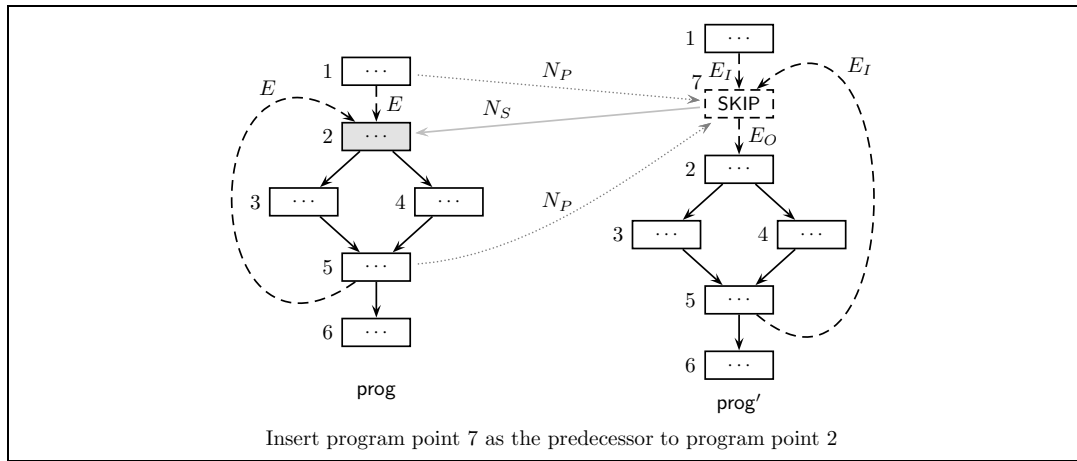


Figure 6. An example of insertion of predecessors transformation

is obtained by inserting the new program point 7 as the predecessor to program point 2. Let us use the ordered sequence $\langle 1, \dots, 6 \rangle$ for indexing matrices associated with **prog**. Let $\text{succs} = \langle 0, 1, 0, 0, 0, 0 \rangle$ represent a set containing program point 2. Let $\text{newpoints} = \langle 0, 0, 0, 0, 0, 0, 1 \rangle$ denote the set of new program points predecessors to succs . The ordered sequence for indexing matrices for **prog'** is $\langle 1, \dots, 6, 7 \rangle$. The new program point 7 is placed at the end of the list.

We model a transformation of the control flow graph of a program by an application of IP as a node addition transformation (Definition 1). Given the arguments of IP, we set up the adjacency matrices for the relations C , N_S , and N_P . For the transformation in Figure 6:

The relation C is represented as the matrix shown here. The rows correspond to program points $1, \dots, 7$ (of the transformed program) and the columns correspond to program points $1, \dots, 6$ (of the input program). Since the new program point does not correspond to any program point in the input graph, the last row has all 0s.

The matrix Succs is a (1×6) matrix which is appended to a (6×6) matrix containing all 0s to get the (7×6) matrix N_S which maps program point 7 of **prog'** to program point 2 of **prog**.

The relation N_P maps program points 1 and 5 (the predecessors of program point 2 in **prog**) to program point 7. Given the vector succs , we identify the adjacency matrix E of the incoming edges to the program points denoted by succs . The matrix N_P is then obtained as $E \cdot N_S$.

1	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	0	0	1
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	1
0	0	0	0	0	0	0

It can be verified that the matrices satisfy the conditions about the nature of the corresponding relations given in Definition 1. For example, the correspondence matrix C denotes a partial (at least one row has all 0s), onto (each column has at least one non-zero

element), and one-to-one relation (each column as well as each row has at most one non-zero element). The adjacency matrix A' of the CFG of prog' can be obtained by substituting these matrices and the adjacency matrix A of the control flow graph of prog in Definition 1.

An insertion of predecessors transformation inserts SKIP statements at the newly inserted program points. If the target of a goto or a conditional statement belongs to the set represented by `succs` then the target is updated to its new predecessor program point (identified using the N_S relation). For the example shown in Figure 6, the target of the conditional statement at program point 5 will be updated to program point 7 in the transformed program. All other statements remain unchanged.

- 4 A *deletion of statements* (DS) transformation deletes a set of program points only if they contain assignment, skip, or goto statements. Program points containing conditional or return statements cannot be deleted. The transformation of the control flow graph is defined as a special case of the node deletion transformation.

The following transformation primitives change only program statements. The control flow of the input program is preserved.

- 5 An *insertion of assignments* (IA) transformation inserts a given assignment statement at a given set of program points.
- 6 A *replacement of expressions* (RE) transformation replaces the occurrences of a given expression at a set of program points by a variable.
- 7 A *replacement of variable operands* (RV) transformation replaces the occurrences of a given variable in the expressions computed at a set of program points by a variable or a constant.

These transformation primitives are sufficient for expressing a large class of compiler optimizations, namely, common subexpression elimination, optimal code motion, loop invariant code motion, lazy code motion, full and partial dead code elimination [13].

4.3. Soundness conditions for semantics preservation

We model semantics preservation of a transformation primitive in terms of a soundness condition. A *soundness condition* defines certain global dataflow properties of a program which guarantee that the program and its transformed version obtained by an application of the primitive are semantically equivalent. We define soundness conditions using a temporal logic called computational tree logic with branching past (CTL_{bp}) [16]. In the following discussion, we do not assume familiarity with CTL_{bp} and explain the notation wherever required.

We first define the notion of semantic equivalence. A store σ denotes the valuations to all variables in a program. The return value of the program is denoted by a variable result. It can take a special value \perp called the error value. A state is a pair (n, σ) of a program point n and the associated store σ . A state transition relation \rightsquigarrow defines how program statements affect the program state. Let the entry point of a program prog be `entry`. A program trace ρ is an infinite sequence of states $s_1 \rightsquigarrow \dots s_n \rightsquigarrow \dots$ where $s_1 = (\text{entry}, \sigma_1)$ is an initial state with σ_1 as an initial store and for all i , $s_i \rightsquigarrow s_{(i+1)}$ according to the statement semantics. A trace ρ

$$\begin{aligned}
\text{Dead_OUT}(v) &= \text{AX}(\text{AW}(\neg(\text{Use}(v)), \text{Def}(v))) \\
\text{Same_Value_IN}(v, x) &= \\
&\text{AY}(\text{AS}(\neg(\text{Def}(v)) * \neg(\text{Def}(x)), \text{Same_Value_OUT_1}(v, x) + \text{Same_Value_OUT_1}(x, v))) \\
\text{EqValue_IN}(v, e) &= \text{AY}(\text{AS}(\text{Transp}(e) * \neg(\text{Def}(v)), \text{AssignStmt}(v, e))) \\
\text{Available_IN}(e) &= \text{AY}(\text{AS}(\text{Transp}(e), \text{Comp}(e))) \\
\text{Anticipatable_OUT}(e) &= \text{AX}(\text{AW}(\text{Transp}(e) \wedge \neg\text{exit}, \text{Antloc}(e)))
\end{aligned}$$

Figure 7. Global program properties as temporal logic formulae

is *terminating* if there exists $i \in \mathcal{N}$ such that $p_i = \odot$. Let $\text{end}(\rho)$ be such that $p_{\text{end}(\rho)} = \odot$ and for all $i \in \mathcal{N}$, $i < \text{end}(\rho)$ implies that $p_i \neq \odot$. The program point \odot does not contain any statement and denotes termination i.e. $(\odot, \sigma) \rightsquigarrow (\odot, \sigma)$.

Definition 2 (Semantic equivalence) Consider a terminating trace ρ of a program prog1 . A program prog2 is semantically equivalent to prog1 if there exists a (unique) terminating trace ρ' of prog2 such that (1) the initial stores are equivalent: $\sigma_1 \triangleright \sigma'_1$ where $\sigma \triangleright \sigma'$ if for every variable v of prog1 , $\llbracket v \rrbracket \sigma = \llbracket v \rrbracket \sigma'$; and (2) $\llbracket \text{result} \rrbracket \sigma_{\text{end}(\rho)} = \llbracket \text{result} \rrbracket \sigma'_{\text{end}(\rho')}$ or $\llbracket \text{result} \rrbracket \sigma_{\text{end}(\rho)} = \perp$.

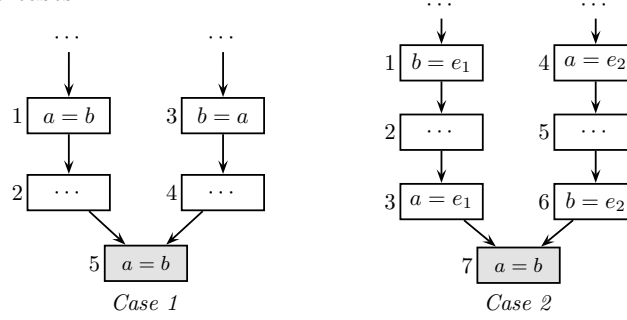
Informally, a program prog2 is semantically equivalent to a program prog1 if starting with equivalent stores, both programs exit normally with the same value of `result` (the return variable) or prog1 aborts, in which case, prog2 may or may not abort. In this definition, we do not make a distinction between non-terminating (diverging) and erroneous traces. The sufficiency of the soundness conditions of a transformation primitive can be proved by induction on the length of a program trace. Here, we shall explain sufficiency of the soundness conditions only informally. For formal proofs of semantics preservation and boolean matrix algebraic semantics of CTL_{bp} operators, we refer the reader to [13].

Deletion of statements

Let prog2 be a program obtained by deleting statements at program points in a set P of a program prog1 . The incoming edges of a program point being deleted are joined to its outgoing edges. To preserve the structure of the CFG, we do not allow deletion of the program entry and exit points. We also do not allow deletion of `ITE` and `RETURN` statements. The deletion of a `SKIP` statement trivially preserves semantics. Note that we are considering transformations of control flow graph representations of programs. For a `GOTO` statement, the target program point of the jump is also its successor in the CFG. Since the incoming edges of a program point are joined with its outgoing edges, the deletion of a `GOTO` statement trivially preserves the structure of the control flow and the program semantics. In the following discussion, we consider semantics preservation for deletion of `ASSIGN` statements.

Suppose $p \in P$ and the statement at p is `ASSIGN`(v, e). We explain the soundness conditions of the DS primitive using the following cases:

1. The expression e is just a variable, say x :
 If the variables v and x are same then it is an assignment of v to itself and deletion of p trivially preserves semantics. Otherwise, the deletion preserves semantics if
 - (a) The variable v is “dead” at all successors of p i.e. it is not used subsequently unless redefined. Or
 - (b) The variable x has the same value as the variable v just before p . We consider the following two cases:



Case 1: Before program point 5, a and b have same value because along all backward paths starting from predecessors of 5, either b is assigned to a or a is assigned to b without any other assignments to a and b in between.

Case 2: Before program point 7, a and b have same value because along all backward paths starting from predecessors of 7, either (i) a is assigned an expression e_1 and before that b is assigned e_1 with no assignment to any operands of e_1 or to a or b in between or (ii) b is assigned an expression e_2 and before that a is assigned e_2 with no assignment to any operands of e_2 or to a or b in between. Additionally, the expression e_1 cannot have the variable b as an operand and the expression e_2 cannot have the variable a as an operand.

The conditions (a) and (b) are respectively given as `Dead_OUT(v)` and `Same_Value_IN(v, x)` in Figure 7. The predicates `Use(v)` and `Def(v)` respectively check if v is used (i.e. appears in an expression) and defined (i.e. appears on LHS) at a program point. The temporal logic operators `AX` and `AW` are forall (universal) successor and weak until operators. `AX(φ)` holds at a program point if φ holds at all of its successors. `AW(φ, ψ)` holds at a program point if along all forward paths φ holds until ψ holds or φ holds forever. `AY` and `AS` are universal predecessor and since operators. These are counter parts of `AX` and `AW` operators for backward paths (also called past operators).

For brevity, we omit the definition of `Same_Value_OUT_1`. In *Case 1*, `Same_Value_OUT_1(a, b)` holds at $\{1\}$ and `Same_Value_OUT_1(b, a)` holds at $\{3\}$. In *Case 2*, `Same_Value_OUT_1(a, b)` holds at $\{3\}$ and `Same_Value_OUT_1(b, a)` holds at $\{6\}$.

2. The expression e is either a constant or a unary or a binary expression.
 The deletion preserves semantics if:
 - (a) The variable v is “dead” at all successors of p . Or
 - (b) The values of v and e are equal just before p . This is possible if along all backward paths starting with the predecessors of p , the expression e is transparent

(i.e. none of operands is defined) and the variable v is not defined until an assignment $\text{ASSIGN}(v, e)$ is encountered. This condition is given as $\text{EqValue.IN}(v, e)$ in Figure 7. The predicate $\text{Transp}(e)$ checks whether e is transparent and the predicate $\text{AssignStmt}(v, e)$ checks whether the statement is $\text{ASSIGN}(v, e)$ at a program point.

Insertion of assignments

Let prog2 be a program obtained by insertion of assignments $\text{ASSIGN}(v, e)$ at program points from a set P of program points of a program prog1 . The control flow graph (CFG) of prog2 is same as that of prog1 . The statements of prog2 are same as the statements of prog1 except for statements at program points in P . Suppose $p \in P$. The insertion of $\text{ASSIGN}(v, e)$ at p preserves semantics if the following conditions are satisfied:

1. The statement at p in prog1 is a SKIP statement and
2. At least one of the following conditions holds:
 - (a) The variable v is not used subsequently unless redefined i.e. $\text{Dead.OUT}(v)$. Or
 - (b) v and e have same value just before p . This ensures that wherever v is used, it has same value in the input and the transformed programs. If e is just a variable, say x , then $\text{Same.Value.IN}(v, x)$ should be satisfied at p . Otherwise $\text{EqValue.IN}(v, e)$ should be satisfied at p . These properties are explained respectively in conditions 1.b and 2.b for deletion of statements primitive

and

3. If e is either a unary or a binary expression, its computation at p in prog2 should not result in computation of a new value along any path. Thus, the expression e should be either “available” or “anticipatable” at p . An expression is available at a program point if it is computed along all backward paths starting from the program point and none of its operands is defined in between and at the point of computation. An expression is anticipatable at a program point if it is computed along all forward paths starting from the program point and none of its operands is defined in between. These conditions are respectively defined as $\text{Available.IN}(e)$ and $\text{Anticipatable.OUT}(e)$ in Figure 7.

The predicate $\text{Comp}(e)$ is satisfied at a program point if the expression e is computed at the program point and none of its operands is assigned. The predicate exit checks if a program point is the program exit. The predicate $\text{Antloc}(e)$ is satisfied at a program point if the expression e is computed at the program.

Replacement of expressions

Let prog2 be a program obtained by replacement of an expression e by a variable v at program points from a set P of program points of a program prog1 . The CFG of prog2 is same as that of prog1 . The statements of prog2 are same as the statements of prog1 except for statements at program points in P . Suppose $p \in P$. The replacement of e by v at program point p preserves semantics if the following conditions are satisfied:

1. The statement at p is an ASSIGN statement with its right-hand side expression as e .
2. The variable v is not an operand of the expression e .
3. v and e should have the same value just before p . If e is just a variable then `Same_Value_IN(v,e)` should be satisfied at p . Otherwise `EqValue_IN(v,e)` should be satisfied at p .

5. Generation of traces for GCC optimizations

The GNU Compiler Collection (GCC) [2] is a widely used and a mature compiler infrastructure. A front-end for GCC is typically generated from a Lex/Yacc specification of the lexical and the syntactic structure of a language. A back-end is generated from a machine description file. However, the optimizers operating on intermediate code are hand-coded. Therefore schemes for validating soundness of GCC optimizers are highly desirable.

GCC uses several intermediate representations, namely, GENERIC abstract-syntax tree, GIMPLE three-address code, static single assignment (SSA), and Register Transfer Language (RTL) [1]. In our validation scheme, we address optimizations of RTL code for programs written in a subset of the C language.

5.1. Processing RTL code

We now explain the RTL representation and the processing required to extract the information relevant for validation. RTL code is organized as an instruction chain (`insn-chain`). It is a doubly linked list of RTL expressions. The following types of RTL expressions (`rtx`'s) are used for representing instructions:

- (1) `insn` is a sequential instruction that cannot jump i.e. cannot pass the control to an instruction other than its successor in the chain.
- (2) `jump_insn` is an instruction that can possibly jump.
- (3) `call_insn` is an instruction that calls a subroutine.
- (4) `barrier` is a marker that indicates that control cannot flow through.
- (5) `code_label` holds a label (string) which is used as a target for jumps.
- (6) `note` contains miscellaneous metadata.

The above `rtx`'s are typically referred to as *insns*. An `insn` in the `insn-chain` contains a unique identifier (a number), the identifier of the preceding `insn`, the identifier of the succeeding `insn`, and an optional instruction `rtx`. An *instruction rtx* is an instruction from the program. In the assembly code generation phase, an instruction `rtx` is pattern matched with target machine dependent `rtx` code. The definitions of all types of RTL expressions are given in `/${GCCHOME}/gcc/rtl.def` file where `/${GCCHOME}` is the base directory of the source code.

Control flow. The control flow of a program is embedded in the `insn-chain`. The instruction `rtx` contained in an `insn` cannot be a jump instruction. Hence, the control flows from the `insn` to its successor `insn`. The `code_label`, `note`, and `barrier` `insns` do not contain instruction `rtx`'s. We therefore do not represent them in the SPOTS representation. A `jump_insn` contains a conditional or an unconditional `goto` instruction `rtx`. A `goto` instruction `rtx` contains a reference


```

(insn 18 13 19 0 (set (reg:CCNO 17 flags)
  (compare:CCNO (reg/v:SI 60 [ j ])
    (const_int 0 [0x0]))) 0 {*cmpsi_ccno_1} (nil) (nil))
(jump_insn 19 18 21 1
  (set (pc) (if_then_else (gt ...) (label_ref 26) (pc))) ...)
(note 21 19 23 1 [bb 1] NOTE_INSN_BASIC_BLOCK)
(insn 23 21 24 1 (set (reg/v:SF 64 [p])
  (div:SF (reg/v:SF 62 [a]) (reg/v:SF 63 [b]))) .. (nil) (nil))
(jump_insn 24 23 25 1 (set (pc) (label_ref 30)) .. (nil) (nil))
(barrier 25 24 26)
(code_label 26 25 27 2 2 "" [1 uses])
(note 27 26 29 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 29 27 30 2 ...)
(code_label 30 29 31 3 4 "" [1 uses])
(note 31 30 33 3 [bb 3] NOTE_INSN_BASIC_BLOCK)
(insn 33 31 34 3 ...)

```

Figure 8. An RTL code fragment

to the `code_label` insn to which the control is to be passed. Since we do not represent `code_label` insns in the SPOTS representation, we take the closure of the control flow from a `code_label` insn to next `insn` or `jump_insn`. As we consider only intraprocedural analyses and optimizations, we do not allow function calls (and hence `call_insns`) in programs.

Figure 8 shows an RTL code fragment for the C program given in Section 2. The insns correspond to the if-else statement in the code. In (insn 18 13 19 ...), the identifier of the insn is 18. For an insn p , the identifier is obtained as `INSN_UID(p)` in the GCC source. Insn 13 (not given here) is the preceding insn in the insn-chain whereas insn 19 is the succeeding insn in the insn-chain. They can be respectively obtained by `PREV_INSN(p)` and `NEXT_INSN(p)` in the GCC source. Figure 9(a) shows the doubly linked insn-chain.

Figure 9(b) shows the control flow of the RTL code fragment. A node p denotes an insn p and an (solid) edge denotes the flow of control between insns. Since insn 18 is an `insn_rtx`, the flow of control falls through to its successor insn 19. This is indicated in Figure 9(b) by edge $\langle 18, 19 \rangle$. Insn 19 is a `jump_insn` and the instruction `rtx_if_then_else` is a conditional goto

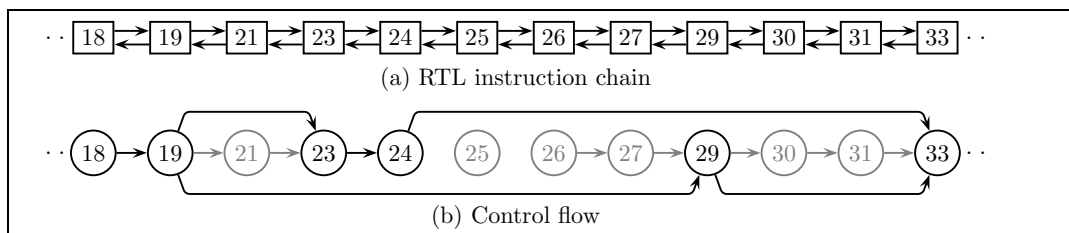


Figure 9. Control flow of the RTL code fragment

```

18 ( ASSIGN @17 ( compare j 0 ) )
19 ( ITE ( gt @17 0 ) ( GOTO 29) )
23 ( ASSIGN p ( div a b ) )
24 ( GOTO 33 )
29 ( ASSIGN q ( div a b ) )
33 ( ASSIGN @17 ( compare m 0 ) )

```

Figure 10. Contents of the RTL code fragment

statement. The target of goto is given as (label_ref 26) rtx indicating that if the condition in the instruction rtx is true then the control goes to insn 26. However, insn 26 is a code_label insn. Hence, we take closure of the flow of control until we reach an insn or a jump_insn. Insn 27 is a note_insn but insn 29 is an insn rtx. Thus, we have edge $\langle 19, 29 \rangle$. If the condition of the goto in insn 19 evaluates to false then it follows edge $\langle 19, 23 \rangle$, insn 21 being a note_insn. Since insn 24 is an unconditional goto, the control cannot fall through to a succeeding insn in the insn chain. This is indicated by the barrier_insn 25.

Contents. Program statements are embedded in insns as instruction rtx's. We eliminate machine specific details and retain only the part that corresponds to the input program. For instance, consider insn 18 in Figure 8. The instruction rtx is a set statement which assigns an expression (compare j 0) to a temporary variable (or register) denoted by number 17. The corresponding instruction is given in Figure 10 where ASSIGN denotes an assignment statement and @17 denotes a temporary variable 17. The prefix “@” is used to distinguish temporary variables generated by GCC from numbers in the input program. The statement at insn 19 is an if-then-else statement which is denoted by ITE. The representation for rest of the statements in Figure 8 is also given in Figure 10.

Printing. GCC uses print_rtl function for printing a list of rtls given the head node of the list (get_insns()). To generate the SPOTS representation, we have written a function print_spots_rtl which is adapted from print_rtl function.

5.2. Generating optimization traces

Given the size and complexity of the GCC code, the task of instrumenting GCC optimizers appears to be daunting. Our approach relies only on traces of optimizing transformations and not on any information about program analyses used by optimizers. We therefore do not have to understand data structures used for computing and storing analysis information and book-keeping operations. This simplifies the task of instrumentation. We can also ignore the fact that GCC uses basic-block level representation whereas our scheme works at the insn level.

Further, program analyses and in particular, profitability heuristics, are the most complex and largest parts of optimizer implementations. On the contrary, optimizing transformation routines are simpler and smaller in size. Figure 11 shows lines of code (LOC) for some optimizer implementations and optimizing transformation routines defined in them. For trace generation, we need to study optimizing transformation routines and other primitive transformation routines used by them. As can be seen from Figure 11, this constitutes only a fraction of

File	≈ LOC (incl. comments)
Entire gcse.c file	6800
Transformation gcse.c/hoist_code	150
Transformation gcse.c/pre_gcse	50
Transformation gcse.c/cprop	15
Entire loop.c file	11900
Transformation loop.c/move_movables	500

Figure 11. Approximate code sizes of GCC v4.1.0 optimizer implementations

the actual code. Our approach is therefore more practical and lightweight than approaches which require an instrumentation of a compiler to generate annotations for target code [29] or to generate proofs of correctness [24].

An optimizing transformation routine is instrumented as follows:

- (1) The SPOTS representation for the input program to the routine is emitted as explained in Section 5.1.
- (2) Flags are set for trace generation. This signals corresponding primitive transformation routines to generate a trace of their execution if invoked.
- (3) The actual transformation routine is executed. The primitive transformation routines called by the code generate traces of their execution.
- (4) Trace generation flags are reset to disable trace generation.
- (5) The SPOTS representation for the optimized program is generated.

An optimizing transformation routine may follow different execution paths for different input programs, possibly calling different primitive transformation routines with different parameters. However, since the body of a primitive transformation routine is instrumented, we do not have to instrument their call sites. Thus, irrespective of the calling context, a primitive always generates a correct trace of its invocation. Further, the instrumentation being merely print statements, is safe and side-effect free.

Figure 12 shows the correspondence of a primitive transformation routine in `cfgrtl.c` and a SPOTS/PVS primitive. The transformation routine `delete_insn` corresponds to the primitive DS. The function `delete_insn` takes as an argument an `insn` to be deleted. We emit the identifier for the `insn` by using `INSN_UID` function. Figure 12 also shows the correspondence between

File	<code>cfgrtl.c</code>
Function	<code>rtl delete_insn (rtl insn)</code>
SPOTS primitive	<code>DS INSN_UID(insn)</code>
File	<code>emit_rtl.c</code>
Function	<code>static rtl emit_insn_after_1 (rtl first, rtl after)</code>
SPOTS primitives	<code>IS INSN_UID(after) INSN_UID(first)</code> <code>IA INSN_UID(first) print_spots_rtl(PATTERN(first))</code>

Figure 12. Examples of primitive transformation routines and corresponding SPOTS primitives

18: @17 = j ? 0		
19: if (@17 > 0) then goto 29		
23: @67 = a / b	IS 38 59	$\left. \begin{array}{l} \text{IP 23 61} \\ \text{IA 61 (ASSIGN @67 (div a b))} \\ \text{RE 23 (div a b) @67} \end{array} \right\} I'_3$
61: p = @67	IA 59 (ASSIGN q @67)	
24: goto 33	DS 38	
29: @67 = a / b	R_LHS 29 @67	$\left. \begin{array}{l} \text{IP 29 60} \\ \text{IA 60 (ASSIGN @67 (div a b))} \\ \text{RE 29 (div a b) @67} \end{array} \right\} I'_2$
60: q = @67	IS 29 60	
33: @17 = m ? i	IA 60 (ASSIGN q @67)	$\left. \begin{array}{l} \text{IS 38 59} \\ \text{IA 59 (ASSIGN q @67)} \\ \text{DS38} \end{array} \right\} I'_1$
34: if (@17 > 0) then goto 41	R_LHS 23 @67	
59: q = @67	IS 23 61	
41: @66 = p + q	IA 61 (ASSIGN p @67)	
48: result = @66		
54: return result		
(a) Optimized program	(b) Generated trace	(c) Equivalent trace

Figure 13. Optimized program and traces under redundancy elimination of GCC

a primitive transformation routine in file `emit_rtl.c` and a sequence of SPOTS primitives. The transformation routine `emit_insn_after_1` inserts first as the successor of `after` in the CFG. The equivalent SPOTS primitive sequence consists of an application of `IS` followed by an insertion of the instruction `rtx PATTERN(insn)`. The textual representation of the instruction `rtx` is emitted by calling `print_spots_rtx` function which is defined in the file `print-spots-rtl.c`.

We have commented the `tree_loop` optimization pass in `passes.c` to enable us to exercise more (RTL) optimizations which otherwise may be performed in the tree optimization phase.

6. Validation of GCC optimizations

6.1. Generating equivalent traces

The primitives appearing in some traces may not satisfy their soundness conditions even if the input and the optimized programs are semantically equivalent. We therefore apply some heuristics to convert a trace to an equivalent one such that the new trace satisfies the soundness conditions of the primitives used in it. Since we check conformance of a trace with actual optimization (commutativity in Figure 3), our scheme is sound though potentially incomplete.

Consider the input program and its CFG shown in Figure 1: (a) and (b). As mentioned in Section 2, the optimized program and the trace in Figure 2 are modified versions since the exact trace does not satisfy the soundness conditions of the transformation primitives. Figure 13(a) shows the actual optimized program. Figure 13(b) shows the generated trace. The second transformation inserts `q = @67` and modifies the reaching definition of `q` at program point 41. Consequently, it does not satisfy the soundness condition of `IA`. Therefore, the validation fails. However, it can be observed that the input and the optimized programs are semantically equivalent even though the intermediate steps are not semantics preserving.

We identify the following issues that prohibit a successful validation:

- (I1) The ordering of the transformations is not appropriate for satisfaction of soundness conditions. For instance, the second transformation IA 59 (ASSIGN q @67) cannot be proven sound unless the variables q and @67 have same value just before insn 59.
- (I2) We do not have any SPOTS transformation primitive corresponding to the transformation R_LHS 29 @67. The transformation replaces the LHS (variable q) of the assignment at insn 29 by variable @67. This is an ad hoc transformation whose soundness cannot be checked independently. It preserves semantics if any of the following conditions holds:
 - (a) The subsequent uses of q are dominated by insn 29 and are also replaced by @67.
 - (b) An assignment q = @67 is inserted immediately after insn 29.

To address I2, we may have to defer the checking of soundness conditions to later transformations in a trace. However, this does not fit well with our compositional validation scheme where a transformation is validated independently from other transformations in the trace. We instead design some heuristics to convert a trace to an equivalent trace for which validation may succeed. Figure 13(c) shows an equivalent trace (same as Figure 2) that satisfies the soundness conditions of the transformation primitives used in it.

Consider the subsequences I_1 , I_2 , and I_3 of the original trace as shown in Figure 13(b). Let I'_1 , I'_2 , and I'_3 be transformation sequences equivalent to I_1 , I_2 , and I_3 respectively with $I'_1 = I_1$. We apply the following two heuristic translations to convert the original trace $\langle I_1, I_2, I_3 \rangle$ to a new trace $\langle I'_3, I'_2, I'_1 \rangle$ shown in Figure 13(c):

- (H1) The application points of the first transformations of the sequences I_1 , I_2 , and I_3 are insns 38, 29, and 23. We apply a heuristic that *statement insertions should be applied to insns in the direction of the control flow*. Therefore, we have two alternatives $\langle I'_3, I'_2, I'_1 \rangle$ and $\langle I'_2, I'_3, I'_1 \rangle$ of which we choose the first sequence.
- (H2) We translate a transformation sequence which pattern matches with sequence I given below to sequence I' .

$$\left. \begin{array}{l} \text{R_LHS } i \times \\ \text{IS } i \text{ j} \\ \text{IA } j (\text{ASSIGN LHS}(i) \times) \end{array} \right\} I \quad I' \left\{ \begin{array}{l} \text{IP } i \text{ j} \\ \text{IA } j (\text{ASSIGN } \times \text{ RHS}(i)) \\ \text{RE } i \text{ RHS}(i) \times \end{array} \right.$$

where $\text{LHS}(i)$ denotes the LHS of insn i and $\text{RHS}(i)$ denotes the RHS of insn i . Note that $\text{LHS}(i)$ and $\text{RHS}(i)$ refer to the values of insn i in the input program to the trace and not to a version of the input program to which the primitive transformation is applied. We translate I_2 to I'_2 and I_3 to I'_3 using this heuristic.

We now explain heuristic H2 with an example. Consider an input program (a) in Figure 14. Program (b) is obtained by transforming (a) by sequence I and program (c) is obtained by transforming (a) by sequence I' . In the SPOTS/PVS framework, we represent a CFG by an adjacency matrix and we do not represent program points (i.e. insn identifiers) explicitly. Clearly, the adjacency matrices of the CFGs and the statement lists of programs (b) and (c) are equal. Thus, sequences I and I' are equivalent with regards to the transformations they perform. For more details on other heuristics, we refer the reader to [13].


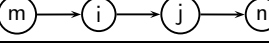
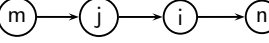
	Index	CFG	Statements
(a)	$\langle m, i, n \rangle$		$\langle -, (\text{ASSIGN } z \ e), - \rangle$
(b)	$\langle m, i, j, n \rangle$		$\langle -, (\text{ASSIGN } x \ e), (\text{ASSIGN } z \ x), - \rangle$
(c)	$\langle m, j, i, n \rangle$		$\langle -, (\text{ASSIGN } x \ e), (\text{ASSIGN } z \ x), - \rangle$

Figure 14. Equivalence of traces

The conversion of generated traces i.e. RTL code for input and optimized programs and the sequence of transformation primitives (ref. Figure 4: “spots to pvs” block) to PVS theories is implemented as AWK and shell scripts.

Suggestions for potential improvements in the GCC code. The generated traces, apart from their use in validation, also give us interesting insights into the functioning and organization of GCC optimizations. In our view, the trace in Figure 13(c) is conceptually more clear than the original trace in Figure 13(b). This suggests a potential reorganization of the GCC code for better understanding and ease of validation. Further, we observe that sequence I_1 in Figure 13(b) transforms a program in a roundabout way. Instead of performing the three transformations, we suggest the following single transformation: RE 38 (div a b) @67 where RE is the primitive for replacement of expressions by a variable. Sequence I_1 and the above transformation both transform statement $q = a/b$ to $q = @67$ except that I_1 does it indirectly. It is also expensive as it involves unnecessary control flow transformations IS and DS. We have also encountered a few other transformation sequences where two or three transformations can be replaced by a single equivalent transformation [13]. These observations suggest scope for potential improvement in the GCC implementation itself.

6.2. Checking conformance of traces

We convert the SPOTS representation of the input and the optimized programs and a trace to a PVS theory. A PVS theory generated from a trace uses transformation primitives defined in the SPOTS/PVS libraries. The definitions in these libraries are operational and mostly written in an executable fragment of the PVS language. However, they contain a few uninterpreted types, namely, *variable*, *constant*, and *operator*. The specifics of these types are not of interest for specification and verification and hence are kept uninterpreted in SPOTS/PVS.

The PVS evaluation environment is a read-eval-print loop that reads expressions from the user, converts them to Common Lisp expressions, evaluates them, and returns the result. It however cannot evaluate uninterpreted symbols. We therefore use theory interpretations [22] to give concrete interpretations to the uninterpreted types. We use the *string* type to give concrete interpretation to the *variable*, *constant*, and *operator* types.

The representations of the programs in the trace are ground terms i.e. do not contain variables and uninterpreted function symbols. From Section 4, we know that transformation primitives are defined in boolean matrix algebra. A trace being a sequence of applications of transformation primitives is a ground term as well.

We then use the PVS ground evaluator to check the conformance of the GCC optimization trace with the actual optimization performed by GCC as follows: The transformations in the trace are applied on the input program. The program thus obtained is then matched with the optimized program generated by GCC (which is also a part of the trace). Both the programs being ground terms, the matching is possible in the PVS ground evaluator. If the two programs match then the trace conforms to the actual optimization performed. This step eliminates the need to trust the instrumentation.

6.3. Checking soundness conditions

The soundness conditions of the transformation primitives are defined in the SPOTS/PVS libraries. As discussed in Section 4.3, the soundness conditions are defined using computational tree logic with branching past (CTL_{bp}). The temporal logic operators are defined using boolean matrix algebra and mu-calculus. Explaining these definitions is beyond the scope of this paper. We refer the reader to [14, 13] for formal semantics of CTL_{bp} operators.

We model check the soundness conditions of the transformation primitives used in the trace on the appropriate versions of the input program. The appropriate version of the input program for an application of a transformation primitive is obtained by simulating the prefix of the trace on the input program. If the soundness conditions of all the primitives used in the trace are satisfied then the trace preserves semantics. Since it is also checked that the trace conforms to the optimization performed (as explained in Section 6.2), we can deduce that the GCC optimization also preserves semantics of the input program.

Using this scheme, we have validated several intraprocedural optimizations of Register Transfer Level (RTL) code in GCC, namely, loop invariant code motion, partial redundancy elimination, lazy code motion, code hoisting, and copy and constant propagation for sample programs written in a subset of the C language.

We also use the framework to check correctness of some analysis information generated by GCC. For instance, we check whether the loops identified by GCC in its loop optimizations are correct with respect to a temporal logic based definition of natural loops.

7. Evaluation of the validation framework

In Section 7.1, we estimate the cost of development of the validation framework in terms of the size of code and PVS specifications. We also discuss how the framework can be extended to other compiler infrastructures and identify the trusted code base of the framework. In Section 7.2, we analyze complexity of the validation approach. We also analyze complexity of the present implementation and evaluate its performance.

7.1. Development cost, extensibility, and trusted code base

Development cost. For estimating the cost of development of the validation framework, in Figure 15 Table (a), we summarize the approximate sizes of C code, AWK scripts, and PVS specifications that form the implementation of the validation framework.

Functionality	\approx LOC	GCC file	\approx LOC
Instrumentation of optimizer routines (C code)	250	<code>gcse.c</code>	6800
Trace to PVS theory (C + AWK code)	900 + 1100	<code>loop.c</code>	5000
SPOTS/PVS libraries (PVS)	1650	<code>rtlanal.c</code>	100
Total	3900	Total	11900

(a) Validation framework

(b) GCC code base

Figure 15. Estimated code and specification sizes

The instrumentation of the source code of GCC consists of 250 lines of C code inserted into optimization and transformation routines. The inserted code fragments simply generate the trace of the optimization being performed. The routines to print simplified RTL representations of programs constitute 900 lines of C code. These are implemented by modifying the GCC print routines for RTL (`print-rtl.c`) and are compiled with the GCC source to form an instrumented GCC executable. A trace of an optimization is converted into a PVS theory. The conversion is implemented as AWK scripts and also includes the heuristics for generation of equivalent traces (ref. Section 6.1). The AWK scripts constitute 1100 lines of code.

The functions used in the PVS theory corresponding to a trace are provided as part of the SPOTS/PVS libraries. The libraries include definitions of boolean matrix operations, temporal logic operators, transformation primitives, and soundness conditions. Together these definitions constitute 1650 lines of PVS specifications.

Figure 15 Table (b) summarizes the sizes of GCC optimization implementations being validated. For the file `gcse.c`, we consider the optimization routines `hoise_code`, `pre_gcse`, and `cprop`. These routines along with the corresponding program analyses constitute most of the code of `gcse.c` which is 6800 lines. For the file `loop.c`, we consider only `move_movables` optimization. The other optimizations, namely, induction variable elimination and strength reduction are not considered presently. We therefore conservatively estimate the relevant code size as 5000 lines (out of 11900 lines of `loop.c`). The `replace_regs` transformation in `rtlanal.c` is used in copy propagation. Thus the total size of the GCC code base (including comments) covered by the validation framework is approximately 11900 lines of code.

The instrumentation thus adds around 21 lines per 1000 lines of GCC implementation. The code size for conversion from traces to PVS theories is proportional to the complexity of the RTL representation. The validation framework requires less than 140 lines of PVS specification per 1000 lines of GCC implementation. Out of the total PVS specifications, around 500 lines are required to define boolean matrix operations and temporal logic operators. These are standard definitions but are not encoded in the predefined PVS prelude theories. The specifications of transformation primitives and their soundness conditions which are specific to our framework constitute around 100 lines per 1000 lines of GCC implementation.

Extensibility. We have already defined a comprehensive set of primitive graph transformations. Graph transformations like node splitting and node merging can be useful in defining optimizations like loop unrolling, splitting, and merging. Thus the validation framework can be extended by adding other primitive program transformations.

To extend the framework, some familiarity with PVS would be required. Note that the process of validation itself does not involve theorem proving. However, typechecking in PVS is not decidable and therefore it may be required to use the prover in order to discharge some type correctness conditions (TCCs).

The only part of the validation framework that is dependent on GCC is the instrumentation code and the conversion of traces to PVS theories. The PVS specifications themselves are completely independent from the GCC implementation. To use the framework with another compiler infrastructure, one therefore needs to instrument the compiler and convert the traces to PVS theories which requires processing the intermediate representation of the compiler.

Trusted code base. The trusted code base (TCB) for a software system is the code on which correctness of the system depends and is distinguished from a much larger code that can be incorrect without affecting correctness of the system (cf. [18]). It is therefore desirable to keep such a code base as small as possible. We now identify the TCB for the validation framework.

Since we check conformance of the trace generated by the instrumented optimizer with the actual optimization, we eliminate the need to trust the instrumentation. Further, the instrumentation being merely print statements is safe and side-effect free. It is also important to note that the primitives are not some reference implementations but are formal definitions. Thus, in order to consider the validation scheme sound, it is not required to implicitly trust the definitions developed by us, as it is possible to argue about their correctness formally. We have also given boolean matrix algebraic semantics to CTL_{bp} operators and we use the PVS ground evaluator for model checking the soundness conditions.

The TCB consists of only the following components: (1) The functions that convert the RTL representation to PVS theories and (2) the PVS ground evaluator. The conversion routines are syntax-directed translators and in general, it is possible to develop enough confidence in them with repeated use or testing. The PVS ground evaluator is part the PVS system which has been used extensively in practice for long and hence qualifies to be called a trusted framework.

7.2. Complexity and performance

The RTL intermediate representation of a program is maintained as a linked list (ref. Section 5.1). The generation of PVS representation for input and optimized programs is possible with a traversal of the list and is thus linear in the size of the program.

The algorithmic technique used for validation is temporal logic model checking. The complexity of model checking computational tree logic (CTL) formulae is linear in both the length of the formula and the size of the Kripke structure [5, 6]. We use a variant of CTL called CTL with branching past (CTL_{bp}). The algorithm for model checking CTL_{bp} formulae is a simple extension of CTL model checking and is also bilinear [16].

The control flow graph of a program is represented as a boolean adjacency matrix. The temporal logic operators and the transformation primitives are defined in boolean matrix algebra and are evaluated using the PVS ground evaluator. These operations involve matrix multiplication. The transformations which do not change the control flow graph take time only linear in the program size whereas the transformations with structural changes are cubic in the program size (due to matrix multiplication). Thus the (worst-case) complexity of the

#rtx	gcc -01				gcc -02 / gcc -03				gcc -0s			
	OPT	<i>K</i>	CC	MC	OPT	<i>K</i>	CC	MC	OPT	<i>K</i>	CC	MC
31	-	-	-	-	cprop	4	0.001	0.07	cprop,hoist	12	0.13	0.59
31	-	-	-	-	pre	9	0.090	0.46	hoist	8	0.09	0.38
35	licm	6	0.04	0.07	sink	4	0.020	0.03	-	-	-	-
48	licm	10	0.23	0.60	pre	15	0.400	1.36	hoist	8	0.20	0.70
80	licm	8	0.19	0.62	licm	11	1.430	4.14	-	-	-	-

Figure 16. Experimental results

conformance checking between the trace and the actual optimization is cubic in the size of the program and linear in the length of the trace. Due to use of matrix multiplication, the complexity of model checking in the PVS based evaluation framework is also cubic in the size of the program and linear in the length of the formula.

Figure 16 summarizes experimental results for validation runs on some sample programs. The column #rtx denotes the number of RTL instructions in the respective programs. The programs are compiled with all optimization settings of GCC, namely, 01, 02, 03, and 0s as shown in the subsequent columns. The optimization flags 02 and 03 resulted in similar traces and hence are combined together. For each optimization setting, four columns: OPT, *K*, CC, and MC are shown. OPT denotes which optimization was applied by GCC. licm, cprop, pre, sink, and hoist respectively denote loop invariant code motion, copy propagation, partial redundancy elimination, code sinking, and code hoisting optimizations.

The column *K* denotes the number of primitive transformation steps in the generated trace. The columns CC and MC report the run-time in seconds for conformance checking of the trace and for model checking soundness conditions, respectively. The typical compilation time by GCC for various optimization settings is in the range of 0.01–0.04 seconds. The run-time is measured on Intel CPU 6600, 2.40GHz, running a Linux distribution.

The optimizer implementations covered during validation are:

- loop invariant code motion (`loop.c/move_movables`),
- partial redundancy elimination (PRE) or global common subexpression elimination (GCSE) through lazy code motion (`gcse.c/pre_gcse`),
- PRE/GCSE through code hoisting (`gcse.c/hoist_code`), and
- copy and constant propagation (`gcse.c/cprop`).

A small number of transformation primitives were enough to cover the generated traces. The traces generated by these 4 optimizer routines were expressible as compositions of only 7 transformation primitives defined by us (ref. Section 4).

The present implementations of conformance checking and model checking are straightforward ground evaluations of the declarative definitions of transformation primitives and temporal logic operators. The PVS ground evaluation environment is a read-eval-print loop that reads expressions from user, converts them to Common Lisp expressions, evaluates them, and returns the results. Though this provides a convenient framework for simulating

specifications, is not very efficient. We expect the present implementation to scale to programs up to several hundred RTXs and traces of a few hundred transformation steps.

In our future work, we plan to consider the following directions to improve scalability of the framework to large programs and traces:

- Efficient implementation of a CTL_{bp} model checker. Note that the actual complexity of model checking is only linear in the program size and hence an optimal implementation of the algorithm should easily scale to programs with several thousand RTXs. A formal correctness proof of the algorithm can be derived in PVS.
- The present implementation translates only a subset of RTL representation to PVS theories. To handle large programs, a comprehensive syntax-directed translator from RTL to PVS should be developed.

8. Related Work

Temporal logic has been used for expressing data flow analyses [27, 26]. Several techniques for specification and verification of optimizations [17, 19] use temporal logic to specify analyses and combine them with rewriting based specifications of program transformations.

The notion of transformation primitives and soundness conditions is suitable for both verification and validation of optimizations. Our earlier specifications in [15] are intended for verification and though equally expressive, are not executable. In [14] we proposed constructive definitions of transformation primitives using boolean matrix algebra and used them in simulation and verification of optimization specifications. In this paper, we have demonstrated that when combined with a trace generation mechanism, the framework of transformation primitives can be used for translation validation of realistic compilers like GCC.

The translation validation approach of Necula [21] tries to discover simulation relations between the input and the optimized programs using some heuristic matching on the respective control flow graphs. Semantic equivalence is then proved by using symbolic evaluation and constraint solving. Some approaches [29, 4] instead require a compiler to generate program annotations as an aid in determining simulation relations. This however may require considerable instrumentation of compiler. In our framework, the compiler is required only to produce traces of optimization runs. As we have discussed in Section 7.1, the cost of instrumentation in our framework is therefore very small.

Goldberg et al. [10] present a proof rule for reasoning about loop optimizations. They develop heuristics to determine which optimizations occurred and synthesize intermediate versions of the input program which may not have been generated by the compiler. While this is similar to our idea of transforming a program step-by-step to get the optimized version of the program, the technique for establishing semantic equivalence is different. The approach by Goldberg et al. uses theorem proving techniques to discharge verification conditions generated by the validation framework. In our framework, the verification conditions are captured by predefined soundness conditions. The validation technique in our approach is therefore temporal logic model checking which is easier and more amenable to automation.

Credible compilers [25] and proof-generating compilers [24] are schemes where compilers themselves generate soundness proofs for each run which are checked by an external proof checker. These approaches expect a lot of work on the part of compilers and also require extensive instrumentation of compilers for this.

The Verifix project [9, 11] proposes use of program checking to ensure correctness of compiler implementations. It checks whether output produced by the compiler meets certain conditions. They have applied it to check front-end implementations. Glesner [8, 7] has introduced the concept of program checking with certificates and has applied it to code selection algorithms. A bottom-up rewrite system specifies valid translations between intermediate code trees and target (machine) code patterns. Given a certificate (sequence of rewrites used by the compiler), the approach recomputes the output independently using the certificate and matches it with compiler's output. This ensures that only valid translations (as specified by the rewrite system) are applied. The traces in our framework can be seen as certificates. However, independent recomputation of the output using the trace only guarantees that the trace is correct but not semantics preserving. In order to prove semantics preservation, we require an additional step which involves model checking soundness conditions of the transformations used in the trace. The use of model checking for translation validation is unique to our approach.

Register allocation is an important back-end activity in a compiler. Huang et al. [12] propose a static analysis for checking correctness of the register allocation phase. The approach involves deriving a mapping between registers in the output program and program variables in the input program. The verification step checks whether the def-use chains in the input program are preserved correctly in the output program for corresponding register allocations.

A complementary approach to verification or validation of optimizing compilers is that of a correct-by-construction mechanism. Leroy [20] presents design of a certified compiler from a subset of the C language to the PowerPC assembly language. The approach uses the Coq theorem prover as a verification and development tool.

As the translation validation frameworks become more practical and exhaustive, the question about correctness of compilers shifts to that of validators themselves. This problem can be tackled by carefully selecting and evaluating the trusted code base of the framework which in our case is the PVS ground evaluator. However, more generally, formal proofs of correctness for validators themselves may be derived. Tristan et al. [28] present formal proofs for validators designed for instruction scheduling optimizations using the Coq theorem prover.

9. Conclusions

We have developed a novel framework for translation validation of GCC optimizers. An optimizer is instrumented to generate a trace of its execution in terms of predefined transformation primitives. If a generated trace conforms to the optimization performed and if the soundness conditions of the primitives used in the trace are satisfied then the optimizer also preserves semantics. The soundness conditions are based on well understood classical data flow analyses and are expressed in a temporal logic. The proof of semantics preservation for a primitive is derived *a priori* and only once whereas the primitive is used in many

optimizations. This simplifies the run-time validation of optimizations. We have validated several intraprocedural bit-vector analysis based optimizations in GCC.

The cost of development of the framework is estimated in terms of code and specification sizes. The instrumentation adds around 21 lines of C code for every 1000 lines of GCC code base considered for validation. The PVS based specifications of transformation primitives and soundness conditions constitute 140 lines for every 1000 lines of GCC code. Thus the cost of development of the framework is reasonable. In particular, the instrumentation is easily achievable even for a complex and large compiler infrastructure like GCC.

In future, we would like to extend the validation framework to SSA-based optimizations. The register allocation and code generation phases of GCC also employ syntactic transformations of programs. It would be interesting to explore whether these transformations can be expressed using primitive transformations and whether soundness conditions for them can be formulated.

In the present implementation, the model checking algorithm is implemented by straightforward ground evaluation in PVS. We plan to implement an efficient model checker in future. A more comprehensive treatment of RTL representation should allow us to achieve scalability to large programs. Presently, the conformance checking is implemented as exact matching between the output obtained by simulation of the trace on the input program and the optimized program generated by the compiler. This can be relaxed to reduce false negatives generated due to mere syntactic mismatch. We have also defined many types of graph transformations like node splitting and merging. These can possibly be used to specify more complex control flow transformations like loop unrolling, merging, and peeling.

Acknowledgments. We wish to thank Supratik Chakraborty for insightful discussions, and Abhijat Vichare and Sameera Deshpande for their help in understanding nuances of GCC. We are also grateful to the anonymous reviewers for their detailed and helpful suggestions.

REFERENCES

1. GCC Internals. <http://gcc.gnu.org/onlinedocs/gccint/>.
2. GCC, the GNU Compiler Collection. <http://gcc.gnu.org/>.
3. Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing, Boston, USA, 1986.
4. C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A translation validator for optimizing compilers. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 291–295, July 2005.
5. Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
6. Edmund M. Clarke, Jr. Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.
7. Sabine Glesner. Program checking with certificates: Separating correctness-critical code. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 758–777. Springer, 2003.
8. Sabine Glesner. Using program checking to ensure the correctness of compiler implementations. *Journal of Universal Computer Science*, 9(3):191–222, 2003.
9. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The Verifix approach. In poster session of CC'96. Technical Report LiTH-IDA-R-96-12, Linkping, Sweden, 1996.

10. B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. In *Proceedings of the Third International Workshop on Compiler Optimization meets Compiler Verification (COCV'04)*, volume 132(1) of *ENTCS*, pages 53–71. Elsevier, May 2005.
11. G. Goos and W. Zimmermann. Verification of compilers. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 201–230. Springer, 1999.
12. Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 281–300. Springer, 2006.
13. Aditya Kanade. *SPOTS: A System for Proving Optimizing Transformations Sound*. PhD thesis, Dept. of Computer Science and Engg., IIT Bombay, 2007.
14. Aditya Kanade, Amitabha Sanyal, and Uday Khedker. A PVS based framework for validating compiler optimizations. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, Sept 2006.
15. Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. Structuring optimizing transformations and proving them sound. *Electr. Notes Theor. Comput. Sci.*, 176(3):79–95, 2007.
16. O. Kupferman and A. Pnueli. Once and for all. In *Proceedings of LICS'95*, pages 25–35, 1995.
17. David Lacey, Neil Jones, Eric Wyk, and Carl Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 283–294, January 2002.
18. Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.
19. Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 220–231, June 2003.
20. Xavier Leroy. Formal verification of an optimizing compiler. In *MEMOCODE*, page 25. IEEE, 2007.
21. George Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Prog. Language Design and Implementation (PLDI'00)*, pages 83–94, June 2000.
22. S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, CSL, SRI International, Menlo Park, CA, April 2001.
23. Sam Owre, Natarajan Shankar, John Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. CSL, SRI International, Menlo Park, CA, September 1999.
24. A. Poetzsch-Heffter and M. Gawkowski. Towards proof generating compilers. In *Proceedings of the Third International Workshop on Compiler Optimization meets Compiler Verification (COCV'04)*, volume 132(1) of *ENTCS*, pages 37–51, May 2005.
25. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, July 1999.
26. D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Proc. of Static Analysis Symposium (SAS'98), Pisa, Italy*, volume 1503 of *Lecture Notes in Computer Science (LNCS)*, pages 351–380, Heidelberg, Germany, September 1998. Springer-Verlag.
27. David Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 38–48, 1998.
28. Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In George C. Necula and Philip Wadler, editors, *POPL*, pages 17–27. ACM, 2008.
29. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A translation validator for optimizing compilers. In *Proceedings of the First International Workshop on Compiler Optimization meets Compiler Verification (COCV'02)*, volume 65(2) of *ENTCS*, 2002.