Published in the Proceedings of the *ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver 2000*, pages 83–95

Translation Validation for an Optimizing Compiler

George C. Necula
University of California, Berkeley
necula@cs.berkeley.edu

Abstract

We describe a translation validation infrastructure for the GNU C compiler. During the compilation the infrastructure compares the intermediate form of the program before and after each compiler pass and verifies the preservation of semantics. We discuss a general framework that the optimizer can use to communicate to the validator what transformations were performed. Our implementation however does not rely on help from the optimizer and it is quite successful by using instead a few heuristics to detect the transformations that take place.

The main message of this paper is that a practical translation validation infrastructure, able to check the correctness of many of the transformations performed by a realistic compiler, can be implemented with about the effort typically required to implement one compiler pass. We demonstrate this in the context of the GNU C compiler for a number of its optimizations while compiling realistic programs such as the compiler itself or the Linux kernel. We believe that the price of such an infrastructure is small considering the qualitative increase in the ability to isolate compilation errors during compiler testing and maintenance.

1 Introduction

Despite a large body of work [CM75, MP67, Mor73, Moo89, You89, WO92] in the area of compiler verification we are still far from being able to prove automatically that a given optimizing compiler always produces target programs that are semantically equivalent to their source versions. But if we cannot prove that a compiler is always correct maybe we can at least check the correctness of each compilation. This observation has inspired the technique of translation validation [PSS98] whose goal is to check the result of each compilation against the source program and thus to detect and pinpoint compilation errors on-the-fly. In this paper we present a few techniques that can be used to implement translation validation and we discuss our initial experience with a translation validation infrastructure for the GNU C optimizing compiler.

One might argue that errors in commercial compilers are the least likely source of headaches for most programmers. While this may be true, compiler manuals contain warnings like the following: "Optimizing compilers can sometimes change your code to something you wouldn't expect. [...] Developers have been able to trace the bad code being generated when optimizations are on by looking at the actual assembly code generated for a function." [Mic99] While this warning suggests a horrifying scenario for an end user, it is, unfortunately, only an accurate description of the state of the art in compiler testing and maintenance. And compiler testing is bound to become more important and more tedious as more demanding architectures are being used as targets.

Our preliminary experience suggests that, with effort similar to that required for implementing one compiler pass, a compiler-development team could build an effective *translation validation infrastructure* (TVI). Such an infrastructure "watches" the compilation as it takes place and points out precisely the mismatches between the semantics of the program being compiled and the semantics of the same program after an individual compiler pass. We have implemented a prototype translation validator for the GNU C compiler. Our prototype is able to handle quite reliably most of the intraprocedural optimizations that `gcc` performs, such as branch optimization, common subexpression elimination, register allocation and code scheduling, while compiling realistic programs such as the compiler itself or the Linux kernel. As an empirical validation of the infrastructure we were able to isolate a known bug in `gcc` version 2.7.2.2. In our experiments the translation validator slows down compilation by a factor of four. In some cases TVI reports errors that are not actual semantic mismatches, but are due to the inability of TVI to understand precisely what transformation took place or why the transformation is correct. We refer to such errors are false alarms. For most optimizations the ratio of false alarms is very low but for some others there is a false alarm in about 10% of the compiled functions. While there is certainly room for improvement in these results and a need for more experimental validation, we believe that translation validation is a promising technique for achieving a qualitative increase in our ability to isolate compilation errors during compiler testing and maintenance, and consequently to

This research was supported in part by the National Science Foundation Grants No. CCR-9875171 and CCR-0081588, NSF Infrastructure Grant No. EIA-9802069, and gifts from AT&T and Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

$$\begin{array}{llll}
\texttt{Instructions} & i & ::= & t \leftarrow E \mid t \leftarrow [E] \mid [E_1] \leftarrow E_2 \mid t \leftarrow \texttt{call}(E, \bar{E}) \mid \texttt{return}(e) \mid \\
 & & & \texttt{label}(L) \mid \texttt{jump}(L) \mid E\ ?\ \texttt{jump}(L_1) : \texttt{jump}(L_2) \\
\texttt{Expressions} & E & ::= & t \mid \&g \mid n \mid E_1\ \texttt{op}\ E_2 \mid \texttt{sel}(M, E) \\
\texttt{Operators} & \texttt{op} & ::= & + \mid - \mid * \mid \& \mid = \mid \neq \mid \leq \mid < \mid \cdots \\
\texttt{Memory} & M & ::= & m \mid \texttt{upd}(M, E_1, E_2) \mid \texttt{updcall}(M, E, \bar{E})
\end{array}$$

Figure 1: The abstract syntax of the IL intermediate language

increase the reliability of our compilers.

Translation validation does not obviate the need for extensive compiler testing suites but is instead a tool for greatly increasing their effectiveness for compiler testing and maintenance. In a traditional compiler test procedure, a test source program is compiled and, if the compiler does not report an internal error, the resulting program is run on a few input values for which the output is known [Cyg]. In most cases, however it is still the task of the tester to inspect manually the output program and to spot subtle compilation errors. For example, exception handling code is both hard to compile correctly and also very hard to test exhaustively. TVI automates this task by comparing the output program to the input program, or to an unoptimized version of the target program, and reporting the exact nature and position of a semantic mismatch, *without* requiring any test cases for the compiled program.

Checking program equivalence is an undecidable problem, and thus we cannot hope to have a complete equivalence checking procedure. However, equivalence checking is possible if the compiler produces additional information to guide the translation validation infrastructure. One contribution of this work is a framework in which such information can be expressed as a *simulation relation*. We show on an example how a checking algorithm would make use of the simulation relation, and we discuss a few guidelines for compiler writers on how to generate such simulation relations.

Typically, each compiler pass transforms the program in a limited way. By looking at the program before and after the transformation, we can hope to detect the transformation that took place, possibly using heuristics or knowledge about the kind of transformations that the compiler performs. A second contribution of this work is a two-step inference algorithm that uses simple heuristics to match the control-flow graphs of the input and output programs and then uses symbolic evaluation along with constraint solving to complete the checking. One way to view this algorithm is as an inferencer for simulation relations.

Some of the advantages of translation validation can be realized with a weaker infrastructure that does not attempt to verify full semantic equivalence but verifies only that the output has certain expected properties. For example, the Touchstone certifying compiler [NL98] proves the type safety of its output when compiling a type-safe subset of the C programming language. Similarly, Special J [CLN+00] does the same for Java, and Popcorn [MCG+99] for yet another type-safe subset of C. In spite of its obvious limitations, this form of result checking has helped with the early discovery of numerous Touchstone bugs, some even in code that was reused from mature compilers. At that time we were asked if one could benefit from the result-checking techniques of Touchstone even in compilers for unsafe languages and for checking more than preservation of type safety, ideally with only minor modifications to the compiler. We believe there is hope to achieve some of these goals, and the current paper describes our initial experience along this path.

In the next section we discuss the equivalence criterion based on simulation relations that out TVI uses. Then we introduce an example program and argue informally that its semantics is preserved by a series of transformations. Starting in Section 4 we formalize this process by showing first the symbolic evaluation pass that collects equivalence constraints, followed in Section 5 by the description of the constraint solver. In Section 6 we report on our preliminary experience with the implementation of our prototype in the context of the GCC compiler.

2 Simulation Relations and the Equivalence Criterion

For exposition purposes we consider that programs are written in the intermediate language (IL) whose syntax is shown in Figure 1. A function body is a sequence of IL instructions. Among instructions we have assignments to temporary registers, memory reads, memory writes, function calls and returns, labels, unconditional jumps and conditional branches. The first argument of a function call denotes the function to be called and $\bar{E}$ denotes a sequence of expressions. (Throughout this paper we are going to use an overbar notation to denote a sequence of elements.) The expression language is also relatively simple containing references to temporaries and to global names, integer literals, and composite expressions using a variety of operators. This language is actually very close to the IL intermediate language that the GNU C compiler uses.

One direction in which IL differs from typical intermediate languages is that the state of memory is represented explicitly. In particular, $\texttt{upd}(M, E_1, E_2)$ denotes the state of memory after a write in previous memory state M at address E_1 of value E_2. And $\texttt{updcall}(M, E, \bar{E})$ denotes the state of memory after a call in memory state M to function E with arguments $\bar{E}$. The expression $\texttt{sel}(M, E)$ denotes the contents of memory address E in memory state M. We will occasionally use variables m to range over memory states.

A *simulation relation* relation between two IL programs S and T (the source and the target) is a set of elements of the form $(\texttt{PC}_S, \texttt{PC}_T, \bar{E})$, where $\texttt{PC}_S$ and $\texttt{PC}_T$ are program points in S and T respectively and $\bar{E}$ is a sequence of boolean expressions referring to temporaries live in S and T at the respective program points. In the current version of our system all such boolean expressions are equalities that contain only temporaries from S on the left-hand side and only temporaries from T on the right-hand side.

Informally, a simulation relation describes under what conditions two program fragments are equivalent. There are quite a few possible equivalence criteria for IL programs. A criterion that is too fine grained (e.g., one that requires the same sequence of memory operations) might prevent possible compiler optimizations. One that is too coarse (e.g., if the source is memory safe then so is the target, such as checked by the Touchstone certifying compiler) does not allow the detection of subtle errors in the optimizer.

In order to define equivalence of program fragments we first define equivalence of a pair of executions. In this work we say that two executions are equivalent if both lead to the same sequence of function calls and returns. Two returns are the same if the returned value and the state of the memory are the same. Two function calls are the same if the state of the memory prior to the call, the arguments and the address of the called function are the same in both cases.

The notion of equivalence of memory states deserves some further discussion because we do not want to constrain it to include the state of memory locations used as spill slots. Such a constraint would not allow TVI to check the operation of the register allocator. We address this issue by assigning names of fresh temporaries to the spill slots followed by rewriting of the program to change spilling instructions into operations using these new temporaries. This effectively undoes the spilling operation. An important added benefit of this operation is that the lack of aliasing between spill slots and other memory locations is made explicit. This renaming operation is actually quite simple since most compilers use only simple indirect addressing from a frame pointer to access the spill slots. In the context of an unsafe language like C it is not possible to guarantee that a location intended as a spill slot will not be changed by a "normal" memory operation. We ignore this possibility, just like any C compiler does, on the grounds that the behavior of such a program is undefined. All we are doing is to adopt the same notion of "undefined" as the compiler does.

Thus, our equivalence criterion is an equivalence of observable behaviors where the set of observable events are function calls and returns. This equivalence criterion is intended to cover all intraprocedural program transformations, such as those performed by the GNU C compiler.[1]

Finally, a simulation relation Σ between S and T is *correct* if for each element $(\mathtt{PC}_S, \mathtt{PC}_T, \bar{E}) \in \Sigma$ *all* pairs of executions of S started at $\mathtt{PC}_S$ and of T started at $\mathtt{PC}_T$, in states that satisfy all boolean conditions $\bar{E}$, are equivalent. Thus, a simulation relation is a witness that two program fragments are equivalent. Note that by this criterion the two program fragments are also required to have the same termination behavior. One can imagine an optimizing compiler producing a simulation relation as a way to explain to an observer how the program changed through the transformation. All that is needed then is a way to check easily such simulation relations. In this paper we go one step ahead and we show that for a moderately-aggressive optimizing compiler like `gcc` it is actually possible to *infer* the simulation relation and thus to avoid modifying the compiler. But in doing this we have to accept that there will be advanced optimizations for which the inference machine will not work, resulting in false compilation errors being reported. We discuss informally in the next section how an example of a simulation relation can be checked and the we go on to formalize this process and describe the inference engine.

3 An Example

Consider the program shown in intermediate form in Figure 2(a) and then again in Figure 2(b) after a few typical transformations. This program writes the values $g * i + 3$ at index i in the array of bytes a, for i running from 0 to $n - 1$. Here a and g are global variables; all of the other variables are locals represented as temporaries in the intermediate language. The program is split into basic blocks labeled b_0 to b_3.

Figure 2(b) shows the same program after a few optimizations. The register r_0 was allocated to hold temporary i, and register r_2 to hold v. The temporary n was spilled to the memory location "$\mathtt{BP} - 4$", where BP denotes the frame pointer register. The loop was inverted and the termination test duplicated. Since i is an induction variable and we assume there is no aliasing[2] between $[\&g]$ and $[\&a + i]$, v is also an induction variable. Finally, the compiler hoists out of the loop (to block b_6) the computation of $[\&g]$ and stores its result in the spill slot "$\mathtt{BP} - 8$".

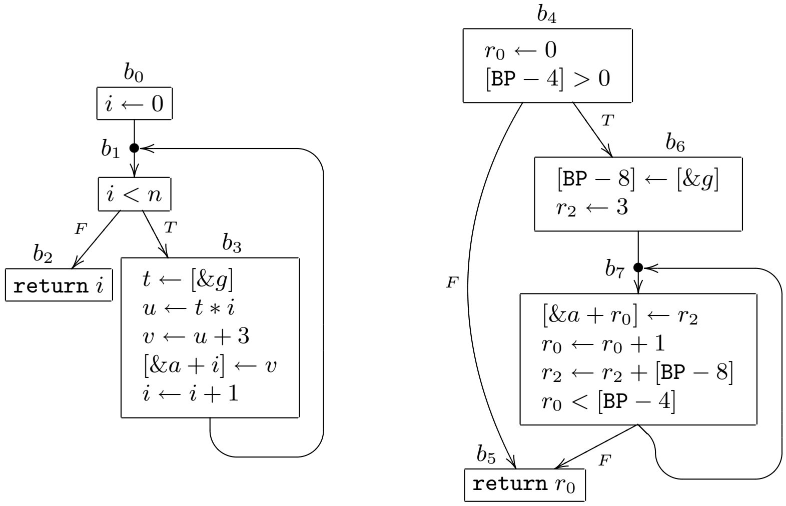
We assume that we are given the simulation relation shown in Figure 2(c) and we have to check its correctness. We omit from this example an additional boolean expression in each element of the simulation relation stating that the memory states should be equivalent in the source and the target at the respective program points. Note that the first element of the simulation relation says that the two programs are equivalent only if they are started in states when "$n = [\mathtt{BP} - 4]$".

Checking equivalence can be accomplished by checking in turn all the elements of the simulation relation. For each row, we examine the definitions of the blocks involved, and we proceed forward in parallel in both the source and the target programs. We stop when we hit a return instruction on both sides, in which case we check that the state of memory and the returned value are the same on both sides, or another pair of related blocks, in which case we check the constraints of this new pair. If all these checks succeed then we can show by an inductive argument that the program fragments are equivalent modulo the constraints from the start row in the equivalence relation. Since the checking process for each element stops when we reach program points related by another element, we can ensure termination of this checking process by requiring that we have "enough" elements in the relation. We will postpone the formalization of this notion until we discuss the inference algorithm.

To illustrate the checking process we show the checking of element 1 (an easy one) and element 5 (the most difficult one). To check element 1, we start by assuming that its

[1] It is possible to extend the set of observable events to include memory writes to the heap. Also, in order to handle correctly the RTL intermediate language of `gcc` we extend the set of observable events to include memory reads and writes to `volatile` locations.

[2] This aliasing assumption could in fact be wrong since the index i is not checked against the array bounds. Nevertheless many C compilers make this assumption.



(a) Original program

(b) Transformed program

1	b_0	b_4	$n = [\mathtt{BP} - 4]$
2	b_1	b_4	$i = 0, n = [\mathtt{BP} - 4]$
3	b_2	b_5	$i = r_0$
4	b_3	b_6	$i = r_0, n = [\mathtt{BP} - 4]$
5	b_3	b_7	$i = r_0, n = [\mathtt{BP} - 4]$ $[\&g] = [\mathtt{BP} - 8]$, $[\&g] * i + 3 = r_2$

(c) Simulation relation

Figure 2: An example IL program before and after a series of transformations, including loop inversion, strength reduction, register allocation with spilling, and instruction scheduling. On the right side we have the simulation relation for this pair of programs.

constraints hold. Then, from the control-flow graphs and the simulation relation, we notice that we need to advance to block b_1 on the source side and stay in place on the target side, in order to hit element 2 of the simulation relation. We now have to check that the constraints of element 2 are met. The first constraint "$i = 0$" is evidently met since we have assigned 0 to i in block b_0. The second constraint is met because we have not modified n or the memory, and we can thus use the assumption directly.

To check element 5, we assume again that its constraints hold, and then we look for pairs of related paths in the source and the target from the current blocks to another pair of related blocks. There are two such pairs: $b_3 - b_1 - b_2$ related to $b_7 - b_5$, and $b_3 - b_1 - b_3$ related to $b_7 - b_7$. (How exactly we discover these paths is explained in Section 5.2.) For both pairs we have to check that they are taken under the same conditions and that they establish the constraints of the related blocks they reach. We show only the checks related to the second pair of paths. We use the primed notation to refer to values of temporaries at the end of the path.

First we have to check the path conditions, i.e., that $i' < n' \equiv r_0' < [\mathtt{BP} - 4]'$, or equivalently that $i + 1 < n \equiv r_0 + 1 < [\mathtt{BP} - 4]'$. Using our assumptions this reduces to proving that $[\mathtt{BP} - 4]' = [\mathtt{BP} - 4]$, which we can be done noting that the memory write from block b_7 cannot change a spill slot. (Recall that we treat spill slots as temporaries and not as memory addresses.)

Now we check the last constraint in element 5, i.e., that $[\&g]' * i' + 3 = r_2'$, or equivalently that $[\&g]' * (i + 1) + 3 = (r_2 + [\mathtt{BP} - 8])$. This is a bit tricky but can be done with knowledge that the compiler does strength reduction. In this case the rule of distributivity of multiplication is used followed by arithmetic simplification to reduce the goal to $[\&g]' * i + [\&g]' + 3 = r_2 + [\mathtt{BP} - 8]$. It remains to prove that $[\&g]' = [\&g]$ which requires arguing that addresses $\&a + i$ and $\&g$ are not aliased. GCC makes this assumption (because it involves addresses of two distinct globals) and so does our TVI. Now we can reduce the goal to $[\&g] * i + [\&g] + 3 = r_2 + [\mathtt{BP} - 8]$ which follows immediately from the assumptions $[\&g] * i + 3 = r_2$ and $[\&g] = [\mathtt{BP} - 8]$.

This example shows that TVI must have similar knowledge of the algebraic rules and aliasing rules that the compiler itself has. A sample of the rules that our TVI uses to check equivalence of boolean expressions appearing in simulation relation elements is shown in Figure 3. We have rules for proving the facts that the compiler itself proves implicitly in the process of optimization: use of assumptions, commutativity of additions, etc. We also have the usual rules saying that equality is a equivalence relation and the congruence rules. The last two rules in this section are used to reason about the contents of memory reads; the first one refers to the contents of a memory location that was just written, and the second one makes used of "cannot alias" information ($\vdash E_1 \neq E_1'$) to prove that a given memory update cannot affect the contents of another memory location. Note that we have no rules for reasoning about the contents of a memory location across a function call. This is one incompleteness in our system motivated by our current focus on intraprocedural optimizations.

In the bottom part of Figure 3 we show rules for reasoning about the equivalence of memory states. For generality, we extend the equivalence judgment to be of the form $\vdash M =_\Delta M'$ to say that the states denoted by M and M' are equivalent, except possibly at those addresses contained in the set Δ. The first two rules in this section are typical substitutivity rules. If the compiler does not reorder memory writes and does not eliminate redundant memory writes then only these two rules are necessary. The other two rules and the Δ annotation are necessary to reason about transformations that change the sequence of memory writes. The

Boolean expression satisfiability: $\vdash E$

$$\frac{}{\vdash E_1 + E_2 = E_2 + E_1} \qquad \frac{}{\vdash E + 0 = E} \qquad \frac{}{\vdash E \geq E} \qquad \frac{\text{p represents the sum of literals n and m}}{\vdash n + m = p}$$

$$\frac{\vdash E_1 = E_1' \quad \vdash E_2 = E_2'}{\vdash \mathtt{sel}(\mathtt{upd}(M, E_1, E_2), E_1') = E_2'} \qquad \frac{\vdash E_1 \neq E_1' \quad \vdash \mathtt{sel}(M, E_1') = E_2'}{\vdash \mathtt{sel}(\mathtt{upd}(M, E_1, E_2), E_1') = E_2'}$$

Memory equivalence: $\vdash M =_\Delta M'$

$$\frac{E_1' \in \Delta \quad \vdash E_1 = E_1' \quad \vdash M =_\Delta M'}{\vdash \mathtt{upd}(M, E_1, E_2) =_\Delta M'} \qquad \frac{\vdash E_2 = \mathtt{sel}(M', E_1) \quad \vdash M =_{\Delta \cup \{E_1\}} M'}{\vdash \mathtt{upd}(M, E_1, E_2) =_\Delta M'}$$

$$\frac{\vdash M =_\Delta M' \quad \vdash E_1 = E_1' \quad \vdash E_2 = E_2'}{\vdash \mathtt{upd}(M, E_1, E_2) =_\Delta \mathtt{upd}(M', E_1', E_2')} \qquad \frac{\vdash E = E' \quad \vdash M =_\Delta M' \quad \vdash \bar{E} = \bar{E}'}{\vdash \mathtt{updcall}(E, M, \bar{E}) =_\Delta \mathtt{updcall}(E', M', \bar{E}')}$$

Figure 3: A few representative rules that define the equivalence checking.

third rule handles the case of "don't care" memory location being written; the fourth one says that M after writing of E_2 to address E_1 is the same as M' if the latter already contains E_2 at that address and is otherwise equivalent to M.

Each of these rules must be designed to match the semantics of the intermediate language. We need enough rules to describe all properties of the IL that the compiler itself uses when transforming the program. This does not mean that TVI must be as complex as the compiler. A compiler is typically more complex because it must also decide which of the rules to use and when. Also, in our implementation of TVI the checker is essentially a pattern matcher with each pattern being a direct transcription of the corresponding logical rule. As a result of this implementation scheme, and also due to one order of magnitude difference in the size of TVI and the compiler, we believe that it is much easier to check by inspection the operation of the translation validator than it is to check the implementation of the compiler.

The attentive reader has noticed that the checking process is quite simple because in fact the difficulty lies in coming up with the simulation relation. The most reliable way to do that is to have it produced by the compiler. However, even for moderately aggressive compilers, such as the GNU C compiler, it is possible to infer the simulation relation, as explained in the next section.

4 Symbolic Evaluation

In this section we start describing the translation validation algorithm in more details. First, a program in the IL form is split into basic blocks, which start at a label and end with a return, a jump or a branch instruction. A core feature of our approach is the use of *symbolic evaluation* to compute the effect of a basic block. To illustrate the major benefit of symbolic evaluation consider the two basic blocks:

```
$t_1 \leftarrow 5;$
$t_2 \leftarrow t_1 + i;$
$t_3 \leftarrow i * t_1;$
$t_4 \leftarrow t_1 + i;$
$t_5 \leftarrow t_4 + t_3;$
return $t_5$
```

```
$t_6 \leftarrow i * 5;$
$t_7 \leftarrow 5 + i;$
$t_8 \leftarrow t_7 + t_6;$
return $t_8$
```

These blocks might seem different if one takes a purely syntactical look at them. However, if we evaluate both of them symbolically we see that they are both equivalent to "`return` $(5 + i) + (i * 5)$". Our point is that symbolic evaluation abstracts over many minor syntactic details such as a permutation of independent instructions (e.g., instruction scheduling), a renaming of local temporaries (e.g., register allocation), or even a change in the order of computing independent subexpressions (e.g., common subexpression elimination). This observation is hardly new; it has appeared before under several disguises such as predicate transformers [Dij76] and value-dependence graphs [WCES94]. But its use for checking correctness of program transformations appears to be new.

A *symbolic evaluation state* S consists of a program point along with a set of symbolic values $\bar{E}$ for the live registers at that point. In our translation validation scheme it is enough to consider only only three kinds of states, corresponding to end-points of the three kinds of blocks, as follows:

$$S ::= \mathtt{ret}(M, E) \quad | \quad b(M, \bar{E}) \quad | \quad E \; ? \; b(M, \bar{E}) : b'(M', \bar{E}')$$

The first form represents a return instruction in memory state M and with returned value E, the second represents a jump to block b in memory state M and with the values of live registers being $\bar{E}$, and the third a conditional branch with guard E and successor blocks b and b' with $\bar{E}$ and $\bar{E}'$ being the values of live registers of b and b' respectively.

For each basic block b, we compute the symbolic state at end of block as a function of the value m of the memory and $\bar{x}$ of the live registers at the block start. For this purpose we create an initial symbolic register state $\rho_b = [\mu \to m, \bar{l}_b \to \bar{x}]$ mapping a special memory register μ and the live registers $\bar{l}_b$ to their initial values. Then we invoke a symbolic evaluation function $SE(\bar{I}(b), \rho_b)$ on the sequence of instructions contained in b and on the initial symbolic register state. The result of the symbolic evaluation function is a symbolic evaluation state possibly depending on the variables m and $\bar{x}$. This allows us to define, for each block b, the evaluation state at the end of the block as a *transfer function* $b(m, \bar{x})$, as follows:

$$b(m, \bar{x}) \stackrel{\text{def}}{=} SE(I(b), \; [\mu \to m, \bar{l}_b \to \bar{x}])$$

$$
\begin{array}{rcl}
\mathtt{SE}(t \leftarrow E; \bar{I}, \rho) & = & \mathtt{SE}(\bar{I}, \rho[t \rightarrow \rho E]) \\
\mathtt{SE}(t \leftarrow [E]; \bar{I}, \rho) & = & \mathtt{SE}(\bar{I}, \rho[t \rightarrow \mathtt{sel}(\rho\mu, \rho E)]) \\
\mathtt{SE}([E_1] \leftarrow E_2; \bar{I}, \rho) & = & \mathtt{SE}(\bar{I}, \rho[\mu \rightarrow \mathtt{upd}(\rho\mu, \rho E_1, \rho E_2)]) \\
\mathtt{SE}(t \leftarrow \mathtt{call}(E, \bar{E}); \bar{I}, \rho) & = & \mathtt{SE}(\bar{I}, \rho[\mu \rightarrow \mathtt{updcall}(\rho\mu, \rho E, \rho\bar{E}), t \rightarrow \mathtt{call}(\rho\mu, \rho E, \rho\bar{E})]) \\
\mathtt{SE}(\mathtt{return}(E), \rho) & = & \mathtt{ret}(\rho\mu, \rho E) \\
\mathtt{SE}(\mathtt{jump}(b), \rho) & = & b(\rho\mu, \rho\bar{l}_b) \\
\mathtt{SE}(E \mathbin{?} \mathtt{jump}(b_1) : \mathtt{jump}(b_2), \rho) & = & \rho E \mathbin{?} b_1(\rho\mu, \rho\bar{l}_{b_1})) : b_2(\rho\mu, \rho\bar{l}_{b_2}))
\end{array}
$$

Figure 4: The symbolic evaluation algorithm

where $I(b)$ denotes the sequence of instructions of block b, and $\bar{l}_b$ is the set of live registers at start of block b.

The symbolic evaluation function *SE* is defined in Figure 4. Depending on the kind of instruction at hand, the symbolic evaluation state is modified accordingly and the function $\mathtt{SE}$ is invoked recursively on the next instruction. We write ρE to denote the expression obtained by applying the substitution ρ to E. We write $\rho[t \rightarrow E]$ to denote the state obtained from ρ after setting t to E. Note that the memory write instruction sets the memory register, and the call instruction modifies both the memory register and the register where the result of the call is placed. The last three cases correspond to terminal instructions and build symbolic states directly. For example, the transfer function of the basic block b_3 from Figure 2(b) is:

$$b_3(m, i, n) \overset{\text{def}}{=} b_1(\mathtt{upd}(m, \&a + i, (\mathtt{sel}(m, \&g) * i) + 3), i + 1, n)$$

and of the basic block b_1 is:

$$b_1(m, i, n) \overset{\text{def}}{=} i < n \mathbin{?} b_2(m, i) : b_3(m, i, n)$$

An alternative view of the symbolic evaluation strategy described here is as a rewriting of the body of an IL function as a purely functional program composed of a series of mutually recursive function definitions (the basic block transfer functions). The benefit is that equivalence of such programs is easier to verify due to lack of side effects. Furthermore, as suggested by the example from the beginning of this section, symbolic evaluation produces syntactically identical transfer functions for many basic blocks that differ in the names of registers or in the order of non-dependent instructions.

Symbolic evaluation simulates symbolically the runtime effects of a sequence of instructions and must model accurately these effects. But since it performs mostly substitution the symbolic evaluation phase does not have to make any decisions based on the semantics of various operators. All such decisions are postponed to checking phase, which is discussed next.

5 Checking Symbolic State Equivalence

We designed two equivalence checking algorithms. One relies on the simulation relation being available (we call this the *checking* algorithm) and another that does not rely on such information (the *inference* algorithm). The checking algorithm is theoretically more powerful and it constitutes both a correctness criterion and a completeness goal for the inference algorithm. In this stage of the project we did not want to modify gcc and we wanted instead to explore how accurately one can infer the simulation relation by just passively observing the compilation. Our current inference algorithm has the following two limitations with respect to the checking algorithm:

- all branches in the target program must correspond to branches in the source program, and
- all constraints in the simulation relation must be equality constraints.

Both of these limitations make our inference algorithm suitable for all but one of the transformations that gcc performs. The exception is loop unrolling, and then only in one of the four versions of unrolling that gcc uses. Since the checking algorithm is a simpler version of the inference algorithm we discuss only the latter here.

The inference algorithm has three components. One component, called *Scan*, walks the source and the target programs in parallel and collects equivalence constraints. Another component, called *Branch*, assists *Scan* in determining whether a branch in the source program was either eliminated, and then which side was retained or if it was copied to the target program in the same form or in reversed form. We are not handling the case when a branch in the target program does not correspond to a branch in the source program. The third component of the inference algorithm, called *Solve*, is invoked last to simplify the set of equivalence constraints produced by *Scan* until none are left. In this case we declare success. Failures can occur in the *Branch* module or in the *Solve* module. In both cases TVI points out the exact nature of a failure so a human can ascertain whether we have uncovered indeed a compiler bug or just an incompleteness issue in our tool. In the latter case we can update the tool or just make a note that validation is know to fail for a certain test case.

5.1 Collecting Constraints

There are three flavors of constraints that we collect in the set $\mathcal{C}$. Expression equivalence constraints $E = E'$ relate two expressions whose values are equal for all substitutions of free variables with values that satisfy $\mathcal{C}$. Memory expression equivalence $M = M'$ relates two memory expressions that denote memory states whose contents coincides.[3] Finally, symbolic state equivalence $S = S'$ relates two states that, for any substitution of their free variables with values that satisfy $\mathcal{C}$, lead to the *same sequence of function calls and returns*. Two function calls or returns are the same if

[3] We actually use a more general form that allows the two memory states to differ on a given set of addresses, as described in Section 3.

they are executed in the same memory state with the same arguments.

The main entry point *Infer* and the *Scan* component are both shown in Figure 5. We use primed notation to refer to entities from the target program. The inference algorithm maintains two lists of pairs of blocks. *Done* contains those pairs of related blocks that have been processed already and *ToDo* those pairs that have been encountered but not yet processed. The latter list is initialized with the start blocks of the source and target respectively. For each pair still to be processed, we create new parameters to stand for the values of live registers on entry and then we invoke the scanner.

The *Scan* procedure takes, in addition to the source and target states, the sequence of IL instructions leading to these states from the start of the current pair of related blocks being processed. *Scan* first follows jumps in both the source and the target (the first two cases of *Scan*) until on both sides we reach either a branch or a return. In the process, the symbolic state and the sequence of leading instructions are accumulated accordingly. Then *Scan* examines the source state, and if that is a return (the third case of *Scan*) it expects a return on the target side as well. `Scan` terminates in this case by adding two constraints to $\mathcal{C}$.

The interesting case is when the source state is a branch. There are four possibilities here, named $\texttt{ELIM}_T$, $\texttt{ELIM}_F$, `EQ`, $\texttt{EQ}_R$, meaning that either the branch was eliminated and the true or the false side was retained, or the branch corresponds to a target branch either in the same direction or reversed. The decision of which case applies is made by the function *Branch* (discussed in Section 5.2) based on the control-flow graph and the sequences of instructions leading to the branch points. In the $\texttt{ELIM}_T$ case the constraint $\neg E = 0$ (where $\neg$ is the negation operator) is added to say that the source branch condition is always true, and scanning continues with the true side. In the `EQ` case the target must also be a branch and a constraint is added saying that the two branches always go in the same direction. The helper function *MarkRelated* adds a state equivalence constraint and then adds the two blocks to the *ToDo* list.

Notice that *Scan* is guaranteed to terminate because it never looks at a pair of blocks more than once. Furthermore, in practice very few blocks (those that have been duplicated) appear in more than one pair. The number of constraints produced in this stage is also relatively small. We typically have one constraint per conditional, two constraints for each return instruction and for each join point we have as many state constraints as there are successors. It is not a surprise that constraint generation, including *Branch*, is done in about 10 seconds on the complete `gcc` sources. Additional constrains are generated during constraint solving but before moving on to that stage we describe briefly the operation of the *Branch* module.

5.2 Navigating Branches

The *Branch* module is invoked when the *Scan* module described before encounters a branch in the source program. The role of the *Branch* module is to discover whether the branch was eliminated or if it was kept in the target program. We describe here an implementation of *Branch* based on heuristics that were found effective in our experiments.

The input of *Branch* consists of the instruction sequences

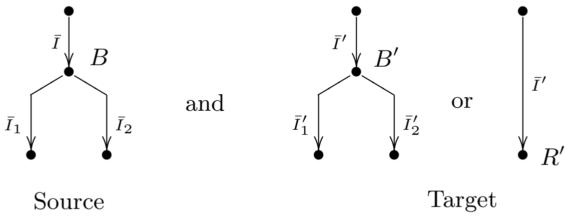
```
Function $Infer(b_0, b'_0)$ is
  $Done \ = \emptyset$
  $ToDo \ = \{(b_0, b'_0)\}$
  $\mathcal{C} = \emptyset$
  while $(b, b') \in ToDo$
    $ToDo \ = ToDo \setminus \{(b, b')\}$
    $Done \ = Done \ \cup \{(b, b')\}$
    create new parameters $m, \bar{x}, m', \bar{x}'$
    $Scan(b(m, \bar{x}), \emptyset, b'(m', \bar{x}'), \emptyset)$
  done
  $Solve(\mathcal{C})$

Function $Scan(S, \bar{I}, S', \bar{I}')$ is
  if $S \equiv b(M, \bar{E})$ and $b(m, \bar{x}) \stackrel{\text{def}}{=} S_1$ then
     $Scan([{}^{M}\!/_{m}, {}^{\bar{E}}\!/_{\bar{x}}]S_1, \bar{I} \cdot I(b), S', \bar{I}')$
  elseif $S' \equiv b'(M', \bar{E}')$ and $b'(m, \bar{x}) \stackrel{\text{def}}{=} S'_1$ then
     $Scan(S, \bar{I}, [{}^{M'}\!/_{m}, {}^{\bar{E}'}\!/_{\bar{x}}]S'_1, \bar{I}' \cdot I(b'))$
  elseif $S \equiv \texttt{ret}(M, E)$ and $S' \equiv \texttt{ret}(M', E')$ then
     $\mathcal{C} = \mathcal{C} \cup \{M = M', E = E'\}$
  elseif $S \equiv E \ ? \ b_1(M_1, \bar{E}_1) : b_2(M_2, \bar{E}_2)$ then
     switch $Branch(\bar{I}, \bar{I}')$ of
     case $\texttt{ELIM}_T$ and $b_1(m, \bar{x}) \stackrel{\text{def}}{=} S_1$
        $\mathcal{C} = \mathcal{C} \cup \{\neg E = 0\}$
        $Scan([{}^{M_1}\!/_{m}, {}^{\bar{E}_1}\!/_{\bar{x}}]S_1, \bar{I} \cdot I(b_1), S', \bar{I}')$
     case $\texttt{ELIM}_F$ and $b_2(m, \bar{x}) \stackrel{\text{def}}{=} S_2$
        $\mathcal{C} = \mathcal{C} \cup \{E = 0\}$
        $Scan([{}^{M_2}\!/_{m}, {}^{\bar{E}_2}\!/_{\bar{x}}]S_2, \bar{I} \cdot I(b_2), S', \bar{I}')$
     case EQ and $S' \equiv E' \ ? \ b'_1(M'_1, \bar{E}'_1) : b'_2(M'_2, \bar{E}'_2)$
        $\mathcal{C} = \mathcal{C} \cup \{E = E'\}$
        $MarkRelated(b_1(M_1, \bar{E}_1), b'_1(M'_1, \bar{E}'_1))$
        $MarkRelated(b_2(M_2, \bar{E}_2), b'_2(M'_2, \bar{E}'_2))$
     case $\texttt{EQ}_R$ and $S' \equiv E' \ ? \ b'_1(M'_1, \bar{E}'_1) : b'_2(M'_2, \bar{E}'_2)$
        $\mathcal{C} = \mathcal{C} \cup \{E = \neg E'\}$
        $MarkRelated(b_1(M_1, \bar{E}_1), b'_2(M'_2, \bar{E}'_2))$
        $MarkRelated(b_2(M_2, \bar{E}_2), b'_1(M'_1, \bar{E}'_1))$
     default
        fail
  else
     fail

Function $MarkRelated(b(M, \bar{E}), b'(M', \bar{E}'))$ is
  $\mathcal{C} = \mathcal{C} \cup \{b(M, \bar{E}) = b'(M', \bar{E}')\}$
  if $(b, b') \notin Done$ then
     $ToDo \ = ToDo \ \cup \{(b, b')\}$
```

Figure 5: The main entry point to the inference function and the `Scan` component.

$\bar{I}$ leading to a branch B in the source and $\bar{I}'$ leading either to a branch B' or to a return R' in the target. First, in order to expose more instructions, we follow jumps after the branches until each side of a branch ends with a return or with another branch. The two possible resulting situations are shown below:



We discuss here only the situation when the target side contains a branch. The other case, of a return, is simpler. If the source branch was preserved then it must correspond to the branch on the target side. The test that we perform is as follows (note that this test is considered to be false if the target side has a return):

$\bar{I} \sim \bar{I}'$ `and`
$((B \sim B'$ `and` $\bar{I}_1 \sim \bar{I}'_1$ `and` $\bar{I}_2 \sim \bar{I}'_2)$
`or`
$(B \sim \neg B'$ `and` $\bar{I}_1 \sim \bar{I}'_2$ `and` $\bar{I}_2 \sim \bar{I}'_1))$

The two similarity operations, between instructions sequences and between conditionals, are defined below. Since these operations are performed by heuristics, their results are not straight booleans but rather scores between 0.0 and 1.0, with "`and`" implemented as multiplication and "`or`" implemented as the maximum operator.

Two boolean conditionals are similar if they could be obtained from each other using simple transformations, such as from $E_1 > E_2$ to $E'_1 > E'_2$ or to $E'_1 < E'_2$. We assign low similarity scores to pairs of conditionals where the comparison operators are not closely related, such as $\geq$ and $\neq$.

To supplement the boolean expression similarity heuristic we also compute instruction sequence similarity, using several elements:

- First we look at the sequence of function calls in the two instruction sequences. For this purpose we consider a return as a call to the function "return" and an indirect function call as a call to the function "indirect". Before doing the comparison, we append as the last element in the sequence of calls, a set of function names that could be called next in *the code following* the instruction sequence. This information is precomputed with an easy fixpoint operation. If the sequence of calls in one instruction sequence is not a prefix of the other's then we cannot have similarity and we assign the lowest score 0.0.

- Next, we look at whether the two instruction sequences lead to points that are already known to be related since they occur in the *Done* or *ToDo* lists. We assign a high score of 1.0 in this case.

The next two heuristics are specific to `gcc` although it is likely that some similar heuristics could be implemented for other compilers.

- Next we look at the sequence of serial numbers of the IL instructions in a sequence. Each IL instruction has a unique serial number. Those instructions that are not eliminated or duplicated across a transformation maintain their serial number, even if their body is modified slightly such as during register allocation. Some instructions are newly introduced in the target, but we can detect that too since their serial numbers are larger than those appearing in the source. We compute the similarity score as the ratio between the number of serial numbers appearing in both instruction sequences divided by the length of the shortest one.

- Finally, in some cases (e.g., when code is duplicated) none of the above heuristics work well enough. In that case we use source line-number information that `gcc` places as special IL instructions in the instruction stream. Since `gcc` is careful to move such information along with the moved instructions (to assist debugging) it becomes a good way to detect code duplication. We assign a score computed like for the sequence of serial numbers.

With these four heuristics we are able to handle reliably all of the transformations performed by `gcc`, except for loop unrolling. The problem there is our current decision to allow only target conditionals that are copies of source conditionals. This problem can be fixed to make the heuristics of *Branch* good enough to handle the control-flow transformations performed by `gcc`. However, unlike other techniques presented in this paper, the *Branch* heuristics are the ones that are most likely to be sensitive to changes in the compiler and also those that might not be easily transferable to other compilers. Perhaps the truly general solution to this problem is for the compiler to annotate branches in the target program indicating how they relate to the source program. This will obviate the need for any heuristics and could greatly improve the robustness of TVI in the face of modifications in the compiler.

5.3 Solving Constraints

The final stage of the inference process solves the constraints collected by *Scan*. There are three kinds of constraints: expression constraints, memory constraints and state constraints. The strategy is to start solving the simplest constraints first. Whenever we find a simple constraint $x = E$ we do several things:

- remove $x = E$ from $\mathcal{C}$

- verify that x does not appear in E. That would lead to circularity that affects the soundness of the algorithm.

- replace in all of $\mathcal{C}$ all occurrences of x by E.

- find out for which pair of basic blocks was the parameter x introduced in *Infer*. Let their transfer functions be $b(m, \bar{x})$ and $b'(m', \bar{x}')$.

- add $x = E$ to the set of boolean expressions contained in the element of the simulation relation corresponding to b and b'

- for all state constraints of the form $b(M, \bar{E}) = b'(M', \bar{E}')$ introduced by *MarkRelated* add to $\mathcal{C}$ the constraint $[m \rightarrow M, \bar{x} \rightarrow \bar{E}, m' \rightarrow M', \bar{x}' \rightarrow \bar{E}'](x = E)$.

Intuitively, the simple constraints are generated first for return instructions. The last step above propagates simple constraints to the predecessors of a block, thus effectively moving them towards the start of the program. When they reach the start they say in what initial states are the pair of programs equivalent.

Assume now that we are in a state without any simple expression constraints. In that case we try to simplify expression and memory constraints using similar algebraic rules to what the compiler itself uses. For example, we take every constraint $E_1 = E_2$ and we compute a canonical form of "$E_1 - E_2 = 0$" by breaking all top-level additions, subtractions and multiplications and rewriting the term as an integer added to a weighted sum of non-arithmetic expressions. For example, we reduce the constraint "$x_1 + 2 * (x_2 + 5) = x_3 + x_2 + x_1 + 20$" to the constraint "$x_2 - x_3 - 10 = 0$", which we can turn into a simple constraint for x_2 or for x_3.

In addition to arithmetic simplification we also simplify memory operations, according to the equation below:

$$\mathtt{sel}(\mathtt{upd}(M, A, E), A') = \begin{cases} \mathtt{sel}(M, A') & \text{if } A - A' \neq 0 \\ E & \text{if } A - A' = 0 \end{cases}$$

The side conditions above are checked using the arithmetic simplifier. If $A - A'$ is simplified to zero then the second choice is used. If it is simplified to a non-zero constant or to an expression involving the difference of the addresses of two globals, then the first choice is used. Otherwise the expression is not simplified.

When we cannot simplify expression constraints anymore, we move on to memory constraints. A memory expression is a sequence of memory updates and function calls. According to our equivalence criterion we do not handle programs in which the compiler moves memory operations across function calls. Thus we can split all memory expressions into segments at function calls. We compare such segments and we add constraints stating, for two corresponding function calls, that the memory states, the functions that are called, and the arguments are all equal. After this step all memory expressions have been reduced to contain just `upd` operations, which we solve using the following rule:

```
$Solve(\mathtt{upd}(M, A, E) = \mathtt{upd}(M', A', E')) =$
    if $A - A' \neq 0$ then
        $\mathcal{C} = \mathcal{C} \cup \{\mathtt{sel}(M', A) = E, \mathtt{sel}(M, A') = E'\}$
    else
        $\mathcal{C} = \mathcal{C} \cup \{M = M', A = A', E = E'\}$
```

where again $A - A' \neq 0$ means that $A - A'$ is simplified to a non-zero constant or to a difference of addresses of globals. These effectively are the aliasing rules that `gcc` uses. The second case is the default one used whenever the compiler does not reorder writes to the heap. Note that this last rule is rather weak in face of sophisticated reordering of memory operations based on aliasing information. Even though this is not currently a problem for `gcc` we plan to improve this aspect of our checker.

Note that since there are a finite number of parameters introduced by *Scan*, the simple-constraint procedure is guaranteed to be executed only a finite number of times. Thus to ensure termination it is sufficient to arrange the simplification procedures to always terminate.

It is instructive to consider in what situations the solver fails. It happens if the simplifier procedures are not as powerful as the compiler in reasoning about expression equivalence, as would be the case with the above simplifier for memory states in the case of a compiler that performs aggressive aliasing analysis. Thus, most of the effort in maintaining a translation validator is spent in the simplifiers.

6 Implementation Details and Early Experimental Results

We implemented the translation validation infrastructure to handle the intermediate language of the GNU C compiler. The `gcc` compiler is a portable optimizing compiler for C and C++ with additional front-ends for Fortran and Java. What makes `gcc` an excellent candidate for experimenting with translation validation is not only the easy availability of source code and documentation but also the fact that it uses a single intermediate language, called RTL, whose intended semantics is also well documented. `gcc` starts by parsing the source code and translating it directly to RTL. During parsing `gcc` performs type checking and even some optimizations such as procedure integration and tail-recursion elimination. The rest of the compiler is rather conventional, composed of a maximum of 13 passes, depending on which optimizations are enabled. Typical transformations are branch optimization, local and global common subexpression elimination, loop unrolling, loop inversion, induction variable optimizations, local and global register allocation and instruction scheduling. Several of the optimizations, such as jump optimization and common subexpression elimination, are run multiple times in different passes.

Access to the RTL programs can be obtained conveniently by instructing `gcc` (with a command-line argument) to produce an easy-to-parse RTL dump after every pass. This does result in large dump files and slows down the compilation considerably but it avoids the need to change GCC.

We implemented the translation validation infrastructure as a standalone program in the Objective CAML dialect of ML. The implementation consists of 10500 lines of code. Of those, about 2000 are the parser and other utility functions that would not be needed for a validator that is integrated with the compiler. The symbolic evaluator is about 1000 lines and the *Solve* module containing the equivalence rules is about 3000 lines. The rest of the implementation are dedicated to auxiliary tasks such as performing liveness analysis and conversion to SSA form. For comparison, the implementation of a typical `gcc` pass is over 10,000 lines of C code. We wrote a shell script that invokes `gcc` with additional command-line flags to request dumps of thirteen IL files (one after each pass), which we parse and compare pairwise. This script is then used as a substitute for the C compiler when building software systems, such as the compiler itself or the Linux kernel. As a general rule we turn on all optimizations supported on the architecture on their most aggressive level.

The implementation follows closely the algorithms described in this paper. We construct the control-flow graph and we perform symbolic evaluation of all blocks. During symbolic evaluation we also keep track of the used and the defined registers, so we can compute liveness information afterwards. As an optimization we are exploring the use of static-single assignment form to reduce the number of independent parameters of basic-block transfer functions and

Program	Functions	RTL instructions	Constraints
GCC 2.91	5909	3,993,552	1,597,420
Linux 2.2	2627	1,021,555	501,108

Table 1: Sizes of the software systems for which we ran translation validation.

thus the number of trivial equality constraints. To prevent excessive use of memory we make sure to share the representation of identical subexpressions.

One of the most expensive operations to be performed by our algorithm is substitution. It is used during the `Scan` module to collect the symbolic state when following jumps and it is also used extensively during the solving procedure. To reduce both the memory usage and the time required to perform the substitution we use the technique of explicit substitutions [ACCL91]. Instead of performing substitutions eagerly we carry the substitution along with the expressions. Later, when we traverse the expression and we hit a reference to a register, we fetch from the accompanying substitution another pair of an expression and a substitution with which we continue the traversal.

We tested our validator on several small examples (a few hundred up to a thousand lines) all of which worked without any problems. (More precisely, we used those examples to find out what is the set of arithmetic and equivalence rules that TVI must be aware of.) For the real test we tried TVI while compiling the `gcc` compiler itself and the Linux kernel. We show in Table 1 the number of functions compiled in each case, along with the total number of IL instructions and a typical sample of the number of constraints arising in checking one pass.

We have several questions that we want to address with our experiments. We know that semantic coverage is very strong by design but we wanted to know how incomplete our system is at the moment. First we measured the effectiveness of the `Branch` module. Then we wanted to know how many of the constraints (each representing a correctness condition at a given program point) cannot be solved in the *Solve* module, as a measure of the incompleteness of our simplifiers. Recall that failures of the *Branch* module to recognize changes in the control-flow graph and failures of the *Solve* module to solve and eliminate constraints translate directly in false alarms that compilation is not correct. Each alarm must be investigated by a human and thus we want to eliminate virtually all false alarms. Finally, we wanted to know how fast the validation process is.

The results for validating the compilation of `gcc` 2.91 are shown in Table 2. The results for the Linux kernel are similar, except that the timings are adjusted proportionally with the number of constraints shown in Table 1. The columns correspond to four optimizations while the lines correspond in order to percentage of failures in the `Branch` module, percentage of unsolved constraints and the time, in minutes, to complete one validation pass of the entire compiler, on a Pentium Pro machine running at 400MHz. The timings do not include the dumping and parsing time for RTL, again in the idea that one would integrate the validator in the compiler. When including those times both compilation and validation are about twice as slow.

GGG v2.91	Branch	CSE	Loop	RA	Sched
Branch miss.	1.2%	1.5%	3.0%	0%	0%
Constr.	1.3%	3.5%	3.2%	0.1%	0.01%
Time(min)	8.4	7.3	17.3	9.9	8.8

Table 2: A synthesis of the false alarm ratios we observe at present and the validation time for compiling `gcc`, for several of the passes.

The early results are promising. The `Branch` module turns out to be quite effective and it fails mostly for the few cases of loop unrolling that it is not yet designed to handle. There are a number of *Branch*-related false alarms when validating the common-subexpression phase. These alarms seem to be due to the *Branch* module not being able to recognize reliably when sequences of adjacent conditionals are eliminated.

The second line in the table shows the percentage of constraints that remain unsimplified after *Solve*. While the simplifier is quite good for the back-end phases we noticed a large number of unsimplified constraints when validating the result of loop unrolling and common-subexpression elimination. Preliminary analysis of the test logs suggests that many of the unsimplified constraints in this case are indeed not simplifiable. The problem is that after a mistaken *Branch* result a number of invalid constraints are generated by *Scan*. We hope therefore that an improvement in the performance of the *Branch* module will bring a large reduction in the ratio of false alarms.

In terms of the running time cost of the TVI we observed on the average that the validation of one compilation phase takes about four times as much as the compilation phase itself. If we also add in the time to dump the RTL files and to parse them into the TVI the time is doubled. We believe that the speed can be drastically improved with a few optimizations in the constraint solver (where almost all of the time is spent). For example, we plan to memoize constraint solving to avoid solving a constraint twice.

As a quick validation of the bug-isolating capabilities of the validator we ran the version 2.7.2.2 of `gcc` on a couple of example programs that were known to exhibit bugs in the register allocator and in loop unrolling. In both cases the bugs surfaced as residual non-simplifiable constraints. This is not surprising since TVI performs such a thorough semantic check. However, the alarm in the unrolling case was accompanied by 3 other false alarms, and this was for a program only 100 lines long. Before we can hope for this infrastructure to become actually used in practice we need to work on the false alarm problem, especially for loop optimizations.

6.1 Related Work

The primary inspiration for this work was the original work on translation validation by Pnueli, Siegel and Singerman [PSS98] and our own work on certifying compilation [NL98, Nec98]. In a sense this project can be seen as tackling the goals laid out in [PSS98] using the symbolic evaluation techniques from [Nec98], in the context of a realistic compiler. Additional innovations were required to handle optimizations like spilling and to overcome the lack of assistance from the optimizer. This is made possible in our case by a few key techniques among which the most important is symbolic evaluation.

The advance of the present work over [PSS98] and other similar work [C$^+$97, Goe97], is that the language involved and the range of optimizations handled here are much more ambitious. In [PSS98] translation validation is applied to the non-optimizing compilation of SIGNAL to C. In [C$^+$97] the compiler translates expressions into a fragment of RTL without loops and function calls. In this case, and also in recent work of Kozen [Koz98], validation is simplified considerably by restricting the optimizations such that each source-level construct is compiled into a given pattern in the target code, and no inter-pattern optimizations are allowed.

The advance over our own previous work, and similar work like TIL [TMC$^+$96] or Popcorn [MCG$^+$99], is that we now attempt to validate full semantic equivalence of the source and target as opposed to checking just the preservation of type safety. This makes the current work more widely applicable, for example to compilers for unsafe languages, and should also make it more effective in isolating compiler bugs.

Also related is the recent work of Rinard and Marinov [RM99, Rin99]. There a compiler produces complete proofs of each transformation and the validator checks the proofs. Our formal framework based on simulation relations appears to be similar in spirit to that used in [RM99]. More fundamentally, our work is different in that it attempts to validate the translation with none or minimal help from the compiler, thus making it more easily applicable to existing compilers. Also, the range of transformations that we handle and the scale of the experiments presented here are more ambitious than in any previous work.

The issue of detecting equivalence of program fragments was studied before in the context of merging program variants [YHR92]. There two programs are considered equivalent if they have isomorphic program representation graphs [HPR88, RR89]. PRG's have some of the features of the symbolic evaluation step we use in that they ignore syntactic details such as the order of unrelated instructions. However, PRG isomorphism is not powerful enough to handle all the optimizations in realistic compilers.

7 Conclusions and Future Work

The main message of this paper is that a practical translation validation infrastructure, able to handle many of the common optimizations in a realistic compiler can be implemented with about the effort typically required to implement one compiler pass, say common subexpression elimination. We demonstrate this in the context of the GNU C compiler for a number of its optimizations. We believe this price is small considering the qualitative increase in the effectiveness of compiler testing and error isolation.

In an ideal world compilers will cooperate with their translation validators by emitting enough information of what happened to the code, in order to take all the guesswork out of the validator, thus simplifying and strengthening it.

Our work is not yet complete. As the experimental data shows we still need to improve the validator in order to reduce the number of false alarms. We also need to take serious action against the running time of the validator. We believe that a significant improvement is possible once we memoize all constraints that were generated and processed.

At the moment we handle only the intermediate phases of `gcc`, ignoring the parser and the code generator. The parser seems is harder to handle but is on the other hand relatively stable. We do intend however to implement translation validation for the x86 and IA64 code generators. In the latter case, we need to extend the infrastructure to handle advanced optimizations that exploit IA64 features such as speculative execution and rotating-register files.

We have not yet implemented the code necessary for validating correct translation of exception handling. That we feel will be one of the strong points of a translation validator because exception handling is both notoriously hard to compile correctly and very hard to test.

We have not yet explored one of the major potential strengths of a translation validator, namely the ability to turn, in theory, a regular compiler into a certifying compiler that also produces proofs pertaining to the target code. More precisely, we can imagine that the simulation relations that we infer can be used as the basis for translating assertions and proofs about the source program to assertions and proofs on the target programs, thus effectively bridging the semantic gap between source and target in a sound way. In particular, one flavor of "proofs" that we could imagine translating are the easily available proofs of well-typedness, with the ultimate effect that we use the `gcc` compiler with the Java front-end to produce provably type-safe native-code components.

Acknowledgments

We would like to thank Jeff Foster, Mihai Budiu, Dawn Song, Hong Wang, Ken Lueh and Mark Wegman for useful comments on this work, Shree Rahul for his help with running the experiments and Christopher Rose for writing the XY-pic Latex package with which the diagrams are drawn.

References

[ACCL91] Martin Abadi, Luca Cardelli, P.-L. Curien, and J.-J Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[C$^+$97] Alessandro Cimatti et al. A provably correct embedded verifier for the certification of safety critical software. In *Computer Aided Verification. 9th International Conference. Proceedings*, pages 202–213. Springer-Verlag, June 1997.

[CLN$^+$00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000.

[CM75] L. M. Chirica and D. F. Martin. An approach to compiler correctness. *ACM SIGPLAN Notices*, 10(6):96–103, June 1975.

[Cyg] Cygnus Solutions. *DejaGnu Testing Framework.* http://www.gnu.org/software/dejagnu/dejagnu.html.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Goe97] Wolfgang Goerigk. Towards rigorous compiler implementation verification. In Rudolf Berghammer, editor, *Proc. of the 1997 Workshop on Programming Languages and Fundamentals of Programming*, pages 118–126, 1997.

[HPR88] Susan Horowitz, Jan Prins, and Tom Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 146–157, San Diego, CA, January 1988.

[Koz98] Dexter Kozen. Efficient code certification. Technical Report TR 98–1661, Cornell University, January 1998.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[Mic99] Microsoft Corporation. *Microsoft Developer Network Library*, March 1999.

[Moo89] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, December 1989.

[Mor73] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the First ACM Symposium on Principles of Programming Languages*, pages 144–152, 1973.

[MP67] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Also available as CMU-CS-98-154.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, June 1998.

[PSS98] Amir Pnueli, M. Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume LNCS 1384, pages 151–166. Springer, 1998.

[Rin99] Martin Rinard. Credible compilers. Technical Report MIT/LCS/TR-776, Massachusetts Institute of Technology, December 1999.

[RM99] Martin Rinard and Darko Marinov. Credible compilation. In *Proceedings of the Run-Time Result Verification Workshop*, July 1999.

[RR89] G. Ramalingam and Thomas Reps. Semantics of program representation graphs. Technical Report CS-TR-89-900, University of Wisconsin, Madison, December 1989.

[TMC+96] David Tarditi, J. Gregory Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *Proceedings of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, January 1994.

[WO92] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1992.

[YHR92] Wuu Yang, Susan Horowitz, and Thomas Reps. A program integration algorithm that accommodates semantics-preserving transformations. *acm Transactions of Software Engineering and Methodology*, 1(3):310–354, July 1992.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.