Translation Validation of Optimizing Compilers by Computational Induction

Amir Pnueli*   Lenore Zuck †   Paritosh K. Pandya‡

August 25, 2000

1 Introduction

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of safety-critical portions of systems. Most verification methods deal with the high-level specification of the system. However, if one is to prove that the high-level specification is correctly implemented at the lower level, one needs to verify the compiler which performs the translations.

Formally verifying a full-fledged optimizing compiler, as one would verify any other large program, is not feasible, due to its size, rates of change, and, possibly, proprietary considerations. Translation validation is a novel approach that offers an alternative to the verification of translators in general and compilers in particular. According to the *translation validation* approach, rather than verifying the compiler itself, one constructs a *validation tool* which, after every run of the compiler, formally confirms that the target code produced on that run is a correct translation of the source program.

Previous work ([PSS98a]) developed such a tool, *CVT*, which managed to automatically verify translations involving about 10,000 lines of source code in about 10 minutes. The success of CVT critically depended on some simplifying assumptions which restricted the source and target to programs with a single external loop, and assumed a very limited set of optimizations.

We offer to remove the limitations assumed in CVT, and achieve a comparable high mark of efficiency for unlimited source programs and maximally optimizing state-of-the-art compilers. Our goal is to develop a methodology, supported by tools, that

*Department of Computer Science, The Weizmann Institute of Science, and Department of Computer Science, New York University, E-mail: amir@wisdom.weizmann.ac.il

†Department of Computer Science, New York University, E-mail: zuck@cs.nyu.edu

‡Computer Science Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Colaba, Mumbai 400 005 India, E-mail: pandya@tcs.tifr.res.in

will provide practical means for ensuring the correct and reliable operation of state-of-the art optimizing compliers. A possible application of this work will be to build instrumentation in the optimizing compiler that will allow it to generate its own proof scripts, which can be checked by a simple theorem prover and yield the translation validation.

Automatic translation validation would be of much value to groups who develop applications in a process in which analysis, verification, and simulation are performed at the source level and it is crucial that whatever integrity has been achieved at the source level is not jeopardized by a faulty translation from source to target. This problem of the compiler being the weak link in an otherwise high-integrity system development process is particularly acute in safety-critical systems, where certification of any used compilers is required by standards, and also in compilation-into-silicon developments, where a translation error may be very expensive to repair. Compiler developers will also benefit from this work, since it will provide them with an extremely effective debugging tool.

In general terms, our strategy is first to give common semantics to the source language of the compiler (assumed to be C) and the target language of the compiler using the formalism of *Transition Systems*. The notion of a target code T being a correct implementation of a source S is that of *refinement*, stating that every computation of T corresponds to some computation of S with matching values of the corresponding variables. In Figure 1 we present the process of refinement as completion of a mapping diagram.

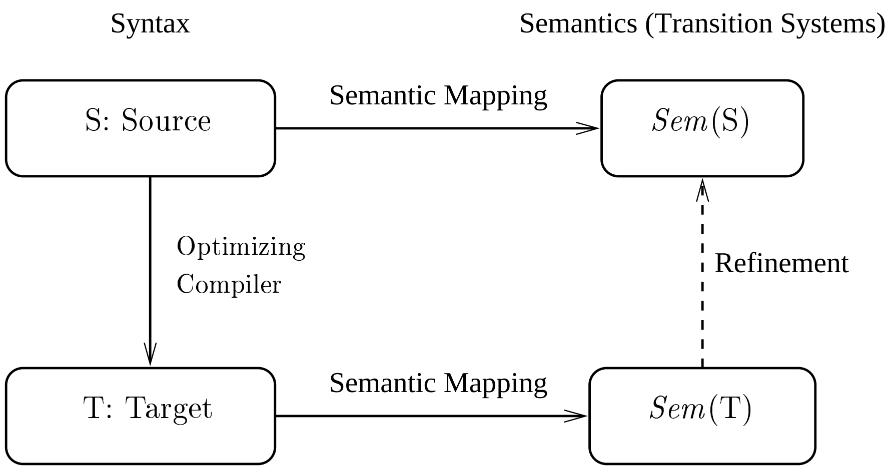

Figure 1: Refinement completes the diagram.

In the case of minor, or no, optimizations, the proof that the target code refines the source program is reduced to the proof of the validity of a set of automatically generated *verification conditions* (proof obligations) which are implications in first order logic. For the simpler case that all optimizations are turned off, all that is required is to establish the validity of the set of these verification conditions. Under the (provably realistic) assumption that only restricted optimization is applied to arithmetic expressions, the proof obligations are in a restricted form of first order logic, called "equational formulae", using uninterpreted functions to represent all arithmetical operations. Recent work ([PRSS99]) has shown the feasibility of building a tool for checking validity of such formulae. This tool is based upon finding small domain instantiations of "equational formulae" and then using BDD-based representation to check for validity.

When the optimization switches are turned on, it is no longer sufficient to use the verification conditions which can be generated automatically. The validation tool will need additional information that can only be provided by the compiler and specifies the optimizing transformations that have been applied in the current translation. A major component of our work plan is to introduce a modest and self-contained instrumentation into optimizing compilers, that will provide this essential information in the form of *program annotation*. This annotation will be processed by the validation tool to form *invariant assertions* at selected control points. The verification conditions will then be augmented by being allowed to use these invariants as assumptions but also by the need to establish that these assertions are indeed invariants.

This paper presents the methodology that will allow the validation tool to use the program annotation in order to construct the verification condition. The main tool we develop is the proof rule VALIDATE that uses the information obtained from the optimizer to construct annotated programs, and then, the verification condition.

The methodology developed here suffices to deal with many classical optimizations ([ASU88]), or "early optimizations" of ([Muc97]). Moreover, if we assume that programs are in SSA-form ([WZ91]), then the methodology covers most of the classical optimizations.

Related Work

The work in [Nec00] also deals with translation validation of optimizing compilers. Unlike the approach we pursue, which entails the compiler supplying information to the validator about the optimizations applied, the work in [Nec00] applies heuristics to the optimized code which attempt to guess the optimizations applied and validate them.

In [ZPG00] we present loop optimizations. The work in [ZPG00] is actually a sequel to the work reported here, where which mainly focuses on non-loop optimizations.

Overview

The paper is organized as follows: In Section 3 we describe the semantic model used to describe both the source and target systems. Section 4 introduces and justifies rule VALIDATE— the proof rule used to establish that target codes refine their source. Section 5 contains examples of the use of VALIDATE through several optimization steps. Section 6 contains a brief review of the method used for automatically verifying the validation conditions that VALIDATE generates. We conclude in Section 7.

2 Our View of Validated Compilation

The compiler receives a *source program* written in some high-level language. It translates the source code into some *Intermediate Representation (IR)* or *intermediate code*. The IR we consider here is "three-address code" (each instruction has at most three arguments – the destination and two sources). The internal representation of IR is in the form of a control graph connecting *basic blocks*, each of which is a single-entry single-exit segment of code. One of the basic blocks is identifies as the *initial block* while another is identified as the *terminal block*. The initial block has no immediate predecessors while the terminal block has no successors.

Consider the source program FACT, where we assume an integer input $n \geq 0$. which

```
for (s=1,i=1; i<=n ; i++) s = s*i;
```

Figure 2: Program FACT

computes in s the factorial of n. As a first step, we translate the `for` loop into explicit tests and incrementations, obtaining the following IR:

```
L0:  s=1; i=1; if !(i<=n) goto L2
L1:  s = s*i; i = i+1; if (i<=n) goto L1
L2:
```

Figure 3: Intermediate Representation of FACT

Here `L0` is the initial block, `L2` is the terminal block, and each label maps to a basic block.

The IR is further transformed to an *Static Single Assignment (SSA) form* ([WZ91]). In SSA form, every use (read) of a variable can originate from a single definition (write) of the variable. To accomplish that, each assignment of a variable is given a unique name, and all uses reached by the assignment are renamed to match that name. For branch and join nodes, a ϕ-*function* is added, of the form $v \leftarrow \phi(u_1, \ldots, u_n)$, where n is the number of control flow predecessors of v. The meaning of such an assignment is that if control reaches from the j^{th} predecessor, then the value assigned to v is u_j. The ϕ functions at a location are executed before the other statements that locations.

Following the example above, an SSA-formed representation of FACT is given in Figure 4.

```
L0:  s1=1; i1=1; if !(i1<=n) goto L2
L1:  s3 = φ(s1,s2);    i3 = φ(i1, i2);
     s2 = s3*i3; i2 = i3+1; if (i2<=n) goto L1
L2:  s4 = φ(s1,s2);    i4 = φ(i1, i2)
```

Figure 4: SSA Representation of FACT

The SSA from has proven to be advantageous for performing many optimizations, mainly because it contains much of the data analysis. Thus, for example, it makes dead code elimination of assignments a purely syntactic test. There are efficient algorithms to transform programs into an SSA form. For example, the algorithm in [CPRW91] is effectively linear in the size of the program. Many contemporary compilers use the SSA form. In this paper we assume that all IR codes are in SSA form.

Once a program is in its SSA form, the compiler applies optimizations to it. Each optimization is then validated by the validator, using the translation validation approach. The validator then either succeeds in validating the optimizations, or presents a counterexample to the claim that the optimization does translate its source code accurately. When all optimizations are applied, the compiler generate the target code of the program. This process is illustrated in Figure 5.

3 Transition Systems

In order to present the formal semantics of C sources and the target language we introduce *transition systems* TS, a variant of the *transition systems* of [PSS98b]. TS is the computational model of the translation validation approach.

A *Transitions System* $S = (V, \Theta, \rho, \mathcal{F})$ is a state machine consisting of V a set of *state variables*, Θ an *initial condition* characterizing the initial states of the system, ρ a *transition relation*, and $\mathcal{F}$ an *termination condition* characterizing the final states of the system. relating a state to its possible successors. The variables are typed, and a *state* of a TS is a type-consistent interpretation of the variables. For a state s and a variable $v \in V$, we denote by $v[s]$ the value s assigns to v. The transition relation is an assertion that relates a state of the TS to its possible successors by referring to both unprimed and primed versions of the variables, where the primed versions refer to the

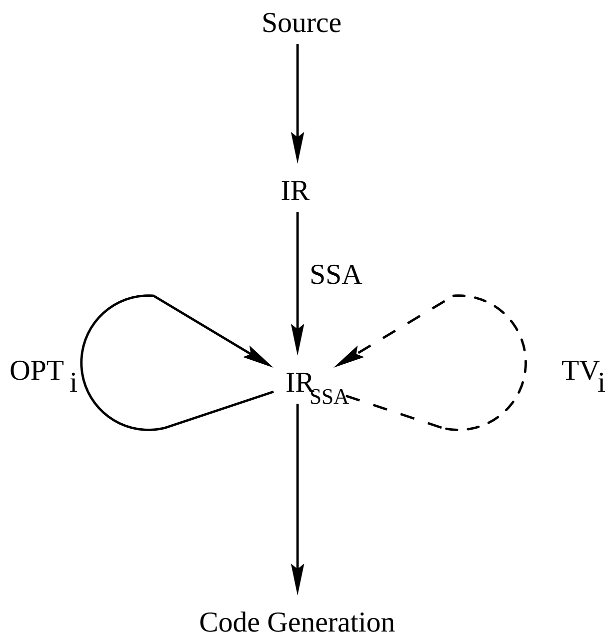

Figure 5: Where Translation Validation Fits in

values of the variables in the successor states, while unprimed versions of variables refer to their value in the originating state. Thus, e.g., a transition may include "$y' = y + 1$" to denote that the value of the variable y in the successor state is greater by one than its value in the old (originating) state.

A computation of a TS is a maximal finite or infinite sequence of states,

$$\sigma : s_0, s_1, \ldots,$$

starting with a state that satisfies the initial condition, i.e., $s_0 \models \Theta$, and every two consecutive states are related by the transitions relation, i.e. $\langle s_i, s_{i+1} \rangle \models \rho$ for every i, $0 \leq i + 1 < |\sigma|$[1]. If σ is finite, then its last state satisfies the termination condition $\mathcal{F}$.

Consider the program FACT in Figure 3. (We choose to use the non-SSA here for simplicity of exposition.) We illustrate how it is transformed into a TS.

The state variables are $V = \{\pi, i, s, n\}$,where π is a control variable (program counter) which points to the next statement to be executed. The range of π is $\{\mathtt{L0}, \mathtt{L1}, \mathtt{L2}\}$. The other variables, i, s and n, are all integers. The initial condition, given by $\Theta\colon \pi = \mathtt{L0}$, states that the program starts at location L0. The termination condition, given by $\mathcal{F}\colon \pi = \mathtt{L2}$, states that the program terminates at location L2.

[1] $|\sigma|$, the *length of* σ, is the number of states in σ. When σ is infinite, its length is ω.

The transition relation ρ can be presented as the disjunction of four disjuncts

$$\rho \quad = \quad \rho_{01} \ \vee \ \rho_{02} \ \vee \ \rho_{11} \ \vee \ \rho_{12},$$

where ρ_{ij} describes all possible moves from `Li` to `Lj`. Each disjunct ρ_{ij} can be further decomposed into

$$\rho_{ij} \quad = \quad (\pi = \mathtt{Li}) \ \wedge \ (\pi' = \mathtt{Lj}) \ \wedge \ \delta_{ij},$$

where the *data transition relation* δ_{ij} describes the transformation of the variables in $V - \{\pi\}$, while the conjunction $(\pi = \mathtt{Li}) \ \wedge \ (\pi' = \mathtt{Lj})$ describes the transformation of the control variable π. *All* TS*s resulting from the translation of programs will have the control variable in their set of state variables.*

We adopt a sequential notation for data relations composed of several successive assignments and tests. In the following table we present the structured representation as well as the explicit form of the four data relations derived for program FACT:

	Sequential	Explicit
δ_{01}	$[\mathtt{s} := \mathtt{1}; \mathtt{i} := \mathtt{1}; !!(\mathtt{i} <= \mathtt{n})?]$	$(s' = 1) \ \wedge \ (i' = 1) \ \wedge \ (n' = n) \ \wedge \ (i' \leq n')$
δ_{02}	$[\mathtt{s} := \mathtt{1}; \mathtt{i} := \mathtt{1}; !(\mathtt{i} <= \mathtt{n})?]$	$(s' = 1) \ \wedge \ (i' = 1) \ \wedge \ (n' = n) \ \wedge \ (i' > n')$
δ_{11}	$[\mathtt{s} := \mathtt{s} * \mathtt{i}; \mathtt{i} := \mathtt{i} + \mathtt{1}; (\mathtt{i} <= \mathtt{n})?]$	$(s' = s \times i) \ \wedge \ (i' = i + 1) \ \wedge \ (n' = n) \ \wedge \ (i' \leq n')$
δ_{12}	$[\mathtt{s} := \mathtt{s} * \mathtt{i}; \mathtt{i} := \mathtt{i} + \mathtt{1}; !(\mathtt{i} <= \mathtt{n})?]$	$(s' = s \times i) \ \wedge \ (i' = i + 1) \ \wedge \ (n' = n) \ \wedge \ (i' > n')$

An example of a computation of the program is:

$$\langle \mathtt{L0}, -, -, 3 \rangle \ \xrightarrow{\rho_{01}} \ \langle \mathtt{L1}, 1, 1, 3 \rangle \ \xrightarrow{\rho_{11}} \ \langle \mathtt{L1}, 2, 1, 3 \rangle \ \xrightarrow{\rho_{11}} \ \langle \mathtt{L1}, 3, 2, 3 \rangle \ \xrightarrow{\rho_{12}} \ \langle \mathtt{L2}, 4, 6, 3 \rangle$$

where each state is described by the values it assigns to $\langle \pi, i, s, n \rangle$.

4 Refinement Mapping

The proof rule used by the validator to establish that target codes are translations of their source codes is VALIDATE, a refinement of the computational induction approach ([Flo67]). The computation induction approach offers a proof methodology to validate that one program *refines* another, by establishing control mapping from target to source locations, data abstraction mapping from target to source variables, and proving that they are preserved with each step of the target program. To deal with optimizations that may change the order of execution of statements, VALIDATE allows for some *annotation*, including invariant assertions and partial data mapping. VALIDATE is described in Figure 6.

1. Establish a *control abstraction* $\kappa: Dom(\mathtt{pc}) \rightarrow Dom(\pi)$ mapping basic blocks into a corresponding value of the source control variable π. The control abstraction should map the initial and terminal blocks into the initial and terminal locations of the source programs.

2. For each basic block `Bi`, form an *invariant* φ_i that may refer to abstract (source) variables.

3. List the variables that are deemed "unnecessary" and verify that they are a subset of the variables eliminated (see notes.)

4. Establish a *partial data abstraction* α_j for every target control location $j \in Dom(\mathtt{pc})$ of the form

$$\alpha_j = \bigwedge_{v \in V^S - \{\pi\}} (v = E_v)$$

assigning to each source state variable v an expression E_v over the state variables or the value $\perp$.

5. For each basic blocks `Bi` and `Bj`, such that `Bj` is a successor of `Bi` in the IR-graph, form the verification condition

$$C_{ij}: \qquad \varphi_i \ \wedge \ \alpha_i \ \wedge \ \delta^T_{ij} \ \wedge \ \delta^S_{\kappa(i),\kappa(j)} \quad \rightarrow \quad \alpha'_j \ \wedge \ \varphi'_j.$$

Figure 6: VALIDATE–A procedure to validate translations

Notes on VALIDATE:

- When we take the invariant assertions of step (2) to be all trivially *true*, and ignore step (3) the proof rule becomes the traditional Floyd-style proof rule [Flo67]. Steps (2) and (3) are included to deal with effects of optimizations.

- The assertions φs in step (2) are program annotations that are expected to be provided by the compiler. Thus, φ_i is used as a hypothesis at the antecedent of the implication C_{ij}. In return, the validator also has to establish that φ_j holds after the transition. Thus, we do not fully trust the annotation provided by the instrumented compiler but, as part of the verification effort, we confirm that the proposed assertions are indeed inductive, and hold whenever visited.

- When source variables are never used (read), they becomes "unnecessary". Since we assume that the programs are in SSA form, an elimination of an assignment implies a removal of a variable. The validator has to make sure that variables that are eliminated are unnecessary, which a the simple syntactic test in step (3).

- The data abstraction mapping in (4) is often independent of the control variables. We then omit the target control subscript from α. For a target state t and source state s, we write $s = \alpha(t)$ to denote the fact that $\pi[s] = \kappa(\mathtt{pc}[t])$ and $v[s] = E_v[t]$ for every $v \in V - \{\pi\}$. Hence, we allow the data abstraction to be partial.

Examples of applications of VALIDATE are presxnted in Section 5.

The Proof Method is Sound:

We provide an argument in support of the claim that the described proof method is sound. That is, if all generated verification conditions are valid, then T is a correct translation of S.

We should realize that the official TS-semantics of a program, such as program FACT, is based on a fully labeled version of the program. A fully labeled version of program FACT is in Figure 7.

```
L0 : s=1;
L0a: i=1;
L0b: if !(i<=n) goto L2
L1 : s = s*i;
L1a: i = i+1;
L1b: if (i<=n) goto L1
L2 :
```

Figure 7: Fully Labeled Representation of FACT

Given such a presentation, we assign a transition relation to each labeled statement which is either a single assignment or a test followed by an appropriate change of control. In the analysis leading to the proof method, we selected a subset of the locations $\mathcal{C}$ (e.g. $\{\mathtt{L0},\mathtt{L1},\mathtt{L2}\}$) which form a *cut-set* in the sense of Floyd [Flo67]. That is, the initial and terminal locations of the program belong to $\mathcal{C}$ and every loop in the control graph of the program contains at least one location from $\mathcal{C}$. Given such a cut-set, we can form the set of *compressed transition relations* which contains a transition relation ρ_{ij} and a corresponding data relation δ_{ij} for every $\mathtt{Li},\mathtt{Lj} \in \mathcal{C}$ such that there exists a control path from Li to Lj not encountering any other location of $\mathcal{C}$ on the way. For a control path

$\mathtt{Li}, \mathtt{Lk_1}, \ldots, \mathtt{Lk_m}, \mathtt{Lj}$, the compressed relation ρ_{ij} (and similarly δ_{ij}) can be computed as the composition $\rho_{i,k_1} \circ \cdots \circ \rho_{k_m,j}$. For example,

$$\delta_{1,1} = \delta_{1,1_a} \circ \delta_{1_a,1_b} \circ \delta_{1_b,1} = (i' = i+1) \ \wedge \ (s' = s \times i) \ \wedge \ (n' = n) \ \wedge \ (i' \leq n').$$

In a similar way, the basic block header locations $\{\mathtt{B4}, \mathtt{B5}, \mathtt{B6}\}$ form a cut-set among the control locations of the target code. It follows that if the pair of states s and s' satisfy ρ_{ij}, there exists a computations segment $s = s_i, s_{k_1}, \ldots, s_{k_m}, s_j = s'$ leading from `Li` to `Lj` along the basic path $\mathtt{Li}, \mathtt{Lk_1}, \ldots, \mathtt{Lk_m}, \mathtt{Lj}$.

Let $t_1, t_2, \ldots, t_n$ be a computation of the target T. According to an observation made in [Flo67], such a sequence can be decomposed into a concatenation of basic paths

$$[t_{m_1}, \ldots, t_{m_2}], [t_{m_2}, \ldots, t_{m_3}], \ldots, [t_{m_{r-1}}, \ldots, t_{m_r}]$$

such that $m_1 = 1$, $m_r = n$, and $\mathtt{pc}[t_1], \mathtt{pc}[t_{m_1}], \ldots, \mathtt{pc}[t_{m_r}], \mathtt{pc}[t_n]$ all belong to the cut-set of T.

Corresponding to the T-sequence $t_{m_1}, t_{m_2}, \ldots, t_{m_r}$, we construct the S sequence

$$s_{m_1}, s_{m_2}, \ldots, s_{m_r}, \text{ where } s_j = \alpha(t_j) \text{ for every } j \in \{m_1, m_2, \ldots, m_r\}.$$

According to the definition of the compressed data relations for T, it follows that $\langle t_{m_j}, t_{m_{j+1}} \rangle \models \delta^T_{m_j, m_{j+1}}$ for every $j = 1, \ldots, r-1$. From the validity of the verification conditions and the fact that the S sequence is α-related to the T sequence, we conclude that $\langle s_{m_j}, s_{m_{j+1}} \rangle \models \delta^S_{m_j, m_{j+1}}$ for every $j = 1, \ldots, r-1$. Therefore, the S sequence can be completed into a full computation which yields on termination a final state s_n that is α-related to the final target state t_n. This establishes that T is a correct translation of system S.

5 Examples of Applications of Validate

Consider the program in Figure 8, and its IR version is Figure 9.

```
i=1; s=1; n=4;
while (i<=n) {s = s*i;  i=i+1;   j=2*n}
```

Figure 8: Source Program

We consider a series of target codes for this program, presented in as IR-to-IR transformations. When discussing each transformation, we distinguish the target variable v from its source counterpart by denoting the target as $\check{v}$.

```
L0:  i_1=1; s_1=1; n=4; if !(i_1<=n) goto L2
L1:  s_3 = φ(s_1,s_2);    i_3 = φ(i_1,i_2);
     s_2 = s_3*i_3; i_2 = i_3+1; j=2*n; if (i_2<=n) goto L1
L2:  s_4 = φ(s_1,s_2);    i_4 = φ(i_1,i_2);
```

Figure 9: Program in IR (SSA)

Constant Propagation and Folding within a Basic Block Let us start with the simple case of *propagation* of constants within a basic block. The constants propagated here are $i_1 = 1$ and $n = 4$ within the first block. The test $!(i_1 \leq n)$ then becomes $\neg(1 \leq 4)$, which is trivially false, and is therefore removed. The resulting code is in Figure 10.

```
L0:  i_1=1; s_1=1; n=4;
L1:  s_3 = φ(s_1,s_2);    i_3 = φ(i_1,i_2);
     s_2 = s_3*i_3; i_2 = i_3+1; j=2*n; if (i_2<=n) goto L1
L2:  s_4 = φ(s_1,s_2);    i_4 = φ(i_1,i_2);
```

Figure 10: Program After Constant Propagation and Folding (within a BB)

We take the data abstraction mapping α_0 to assign all source variables to $\perp$, and α_1 and α_2 to map each source variable to its target counterpart. Verifying the correctness of this translation can be carried out by taking all the invariants (of step (2) of VALIDATE) to be trivially *true*, and ignoring step (3).

Note that the verification condition C_{02} it trivially true since δ^T_{02} is false. The verification condition C_{01}, e.g., is given by:

$$\alpha_0 \wedge \underbrace{\check{i_1}' = 1 \wedge \check{s_1}' = 1 \wedge \check{n}' = 4}_{\delta^T_{01}} \wedge \underbrace{i_1' = 1 \wedge s_1' = 1 \wedge n' = 4 \wedge i_1' \leq n'}_{\delta^S_{01}} \to \underbrace{i_1' = \check{i_1}' \wedge s_1' = \check{s_1}' \wedge n' = \check{n}'}_{\alpha_1'},$$

which obviously holds.

For reasons of clarity, we omitted explicit mention of i_2, i_3, i_4, s_2, s_3 and s_4 in the verification condition since they are not altered by either transitions.

Constant Propagation between Blocks Next, we consider an optimization in which the compiler detected that n, i_1, and s_1 are constants and propagated their values forward into the second block, simplifying the expression `2*4` into `8` and replacing the test in the second block. Similarly, the compiler may propagate the values of i_1 and s_1 into the third block, or, better still, notice that control to L2 comes always from L1, and replace the ϕ functions there by s_2 and i_2 respectively. This leads to the translation of Figure 11.

```
L0:  i₁=1; s₁=1; n=4;
L1:  s₃ = φ(1,s₂);    i₃ = φ(1,i₂);
     s₂ = s₃*i₃; i₂=i₃+1; j=8; if (i₂<=4) goto L1
L2:  s₄=s₂; i₄=i₂
```

Figure 11: Program After Constant Propagation and Folding (between BBs)

For this translation, we maintain the same α as in the previous case. However, without further ado, we obtain the verification condition $C_{11} : \alpha_1 \ \wedge \ \delta_{11}^T \ \wedge \ \delta_{11}^S \rightarrow \alpha_1'$, where

$$\begin{aligned}
\delta_{11}^T\!: \quad & \check{s_3}' = \phi(1, \check{s_2}) \ \wedge \ \check{i_3}' = \phi(1, \check{i_2}) \ \wedge \ \check{s_2}' = \check{s_3}' \times \check{i_3}' \ \wedge \ \check{i_2}' = \check{i_3}' + 1 \ \wedge \ \check{j}' = 8 \ \wedge \\
& \quad \check{i_1}' = \check{i_1} \ \wedge \ \check{s_1}' = \check{s_1} \ \wedge \ \check{n}' = \check{n} \ \wedge \ s_4' = s_4 \ \wedge \ i_4' = i_4 \ \wedge \ \check{i_2}' \leq 4 \\
\delta_{11}^S\!: \quad & s_3' = \phi(s_1, s_2) \ \wedge \ i_3' = \phi(i_1, i_2) \ \wedge \ s_2' = s_3' \times i_3' \ \wedge \ i_2' = i_3' + 1 \ \wedge \ j' = 2 \cdot n \ \wedge \\
& \quad i_1' = i_1 \ \wedge \ s_1' = s_1 \ \wedge \ n' = n \ \wedge \ i_2' \leq n'
\end{aligned}$$

which is not valid unless $i_1 = 1$, $s_1 = 1$, and $n = 4$.

To correct the situation, we carry the information about the propagated constants L1 of the target code by utilizing step (2) of the proof rule and taking $\varphi_1 = \{i_1 = 1 \ \wedge \ s_1 = 1 \ \wedge \ n = 4\}$. The annotated version of the program is in Figure 12.

```
L0:          i₁=1; s₁=1; n=4;
L1:   {φ₁}   s₃ = φ(1,s₂);     i₃ = φ(1,i₂);
             s₂ = s₃*i₃; i₂ = i₃+1; j=8; if (i₂<=4) goto L1
L2:          s₄=s₂; i₄=i₂
```

Figure 12: Annotated Program After Constant Propagation and Folding (between BBs)

(The assertions annotating L0 and L2 are *true* and are therefore omitted.) The verification condition C_{11} now becomes:

$$\alpha_1 \ \wedge \ \underbrace{\check{i_1} = 1 \ \wedge \ \check{s_1} = 1 \ \wedge \ \check{n} = 4}_{\varphi_1} \wedge \ \delta_{11}^T \ \wedge \ \delta_{11}^S \ \rightarrow \ \alpha_1' \ \wedge \ \underbrace{\check{i_1}' = 1 \ \wedge \ \check{s_1}' = 1 \ \wedge \ \check{n}' = 4}_{\varphi_1'}$$

which is valid.

Dead Code Elimination Next, the compiler may detect that i_1, s_1, and n are never used and eliminate the two assignment statements in L0, obtaining the code in Figure 13. To see how VALIDATE can be used to validate this optimization, we note

```
L0:
L1:  s3 = φ(1,s2);    i3 = φ(1,i2);
     s2 = s3*i3; i2 = i3+1; j=8; if (i2<=4) goto L1
L2:  s4=s2; i4=i2
```

Figure 13: Program After Dead Code Elimination

that all the removed assignments refer to unnecessary variables thus satisfying step (3).

Note that (1) the assertion in L1 is also removed, since the only variables it refers to are eliminated, and (2) α' of this step is different from α in that it doesn't refer to the eliminated variables.

Loop Invariant Code Motion Finally, the optimizer detects that the statement $j = 8$ is unnecessarily re-executed on each iteration of the loop and may decide to move it out of the loop. This leads to the optimized translation in Figure 14. To validate

```
L0:  j=8
L1:  s3 = φ(1,s2);    i3 = φ(1,i2);
     s2 = s3*i3; i2 = i3+1; if (i2<=4) goto L1
L2:  s4=s2; i4=i2
```

Figure 14: Program After Loop Invariant Code Motion

this translation using VALIDATE, we re-introduce an assertion in L1, stating that $j = 8$. However, the verification condition C_{01} is invalid now, since letting α_1 assign the source's j to the target j, implies that the source's j equals 8 on first entry to L1, which is false.

There are several possible solution to this problem. One is to allow use of the ϕ functions in α. Thus, for example, we can assign j to $\phi(\perp, \check{j})$ in α_1. Another solution is to introduce a new auxiliary boolean *relevance* variable rel(j), which is set to *false* at L1, and to *true* at L1. In this case,

$$\alpha_1(j) \;=\; \texttt{if rel(j) then } \check{j} \texttt{ else } \perp.$$

We opt for the first option. The verification condition C_{01} thus becomes:

$$\alpha_0 \;\wedge\; \underbrace{\check{j}' = 8}_{\delta_{01}^T} \;\;\rightarrow\;\; \alpha_1' \;\wedge\; \underbrace{\check{j}' = 8}_{\varphi_1},$$

where when evaluating α'_1, $\phi(\perp, \check{j}) = \perp$ since `L1` is accessed from `L0`. Thus, C_{01} is trivially true. (Note that when evaluating C_{11}, the definition of j in α_1 becomes exactly $\check{j}$.)

6 Validating the Verification Conditions

Following the generation of the verification conditions whose validity implies that the target T is a correct translation of the source program S, it only remains to check that these implications are indeed valid. The approach promoted here will make sense only if this validation (as well as the preceding steps of conditions' generation) can be done in a fully automatic manner, with no user intervention.

For performing this validation task, we rely on the efficient decision procedure package developed as part of the CVT tool for the *Sacres* project [PRSS99]. We will illustrate the operation of this procedure on the example formula:

$$u_1 = x_1 + y_1 \ \wedge \ u_2 = x_2 + y_2 \ \wedge \ z = u_1 u_2 \quad \rightarrow \quad z = (x_1 + y_1)(x_2 + y_2)$$

Step 1. Abstract into uninterpreted functions We replace each concrete arithmetic operation by a fresh function symbol. Applied to our example formula, this yields

$$u_1 = F(x_1, y_1) \ \wedge \ u_2 = F(x_2, y_2) \ \wedge \ z = G(u_1, u_2) \quad \rightarrow \quad z = G(F(x_1, y_1), F(x_2, y_2)).$$

where $+$ and $\times$ have been replaced by F and G, respectively. Note that this transformation *strengthens* the statement to be proved. We validate the translation not only for the particular operations $+$ and $*$ *but for all functions F and G.*

Step 2. Eliminate all function symbols Replace each application of a function by a fresh variable and add conjuncts ensuring the functionality of these applications. Applied to our example formula, this yields

$$\left(\begin{array}{ll} & (x_1 = x_2 \wedge y_1 = y_2 \rightarrow f_1 = f_2) \\ \wedge & (u_1 = f_1 \wedge u_2 = f_2 \rightarrow g_1 = g_2) \end{array} \right) \ \wedge \ u_1 = f_1 \ \wedge \ u_2 = f_2 \ \wedge \ z = g_1 \quad \rightarrow \quad z = g_2.$$

Note that since the variable f_1 replaces the term $f(x_1, y_1)$ and f_2 replaces the term $f(x_2, y_2)$ the implication $x_1 = x_2 \wedge y_1 = y_2 \rightarrow f_1 = f_2$ ensures the functionality of the function f. That is, if $x_1 = x_2$ and $y_1 = y_2$ then $f(x_1, y_1) = f(x_2, y_2)$.

Step 3. Find adequate small domains Using the algorithm described in [PRSS99], we assign to each variable of the formula a small integer domain, such that if the formula is valid over these finite domains it is valid in general. For the example formula, our algorithm yields the following domain assignment:

$$\begin{array}{llllll} x_1: \{0\}, & x_2: \{1\}, & y_1: \{2\}, & y_2: \{3\}, & u_1: \{6,7\}, & u_2: \{6,8\}, \\ f_1: \{6\}, & f_2: \{6\}, & g_1: \{4,5\}, & g_2: \{4\}, & z: \{4,5\}. & \end{array}$$

Finally, we submit the formula with the small finite domains to a BDD package to check its validity for the case that the variables range over their respective domains.

Obviously, to handle the more complex situation of compilers with unrestricted source programs and extensive optimizations, we will need to extend the CVT decision package. A clearly necessary addition is the construction of a *simplifying* pre-processor which will perform operations comparable to the optimizing transformations of constant propagation and calculation of constant expressions. Other enhancements are planned in order to deal with arrays and records (C-structures).

7 Conclusion and Future Work

We propose to apply the translation validation approach to state-of-the-art optimizing compilers. This paper lays the foundation for using translation validation for the classical optimizations. In [ZPG00] we extend the approach presented here to deal with loop optimizations.

We are currently working on extending the work reported here to handle instruction scheduling and assembler level array representation. We are also working on an extension of CVT to capture the optimizations discussed here.

We are planning to apply our approach to the validation of optimizing compilers that use the EPIC architecture. In particular, the to Trimaran compiler and SGI's Pro64 compiler.

References

[ASU88] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1988.

[CPRW91] R. Cyron, J. Perrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment from and the control dependece graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 10 1991.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[Nec00] G. Necula. Translation validation of an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000*, pages 83–95, 2000.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In N. Halbwachs and D. Peled, editors, *Proc.* 11*st Intl. Conference on Computer Aided Verification (CAV'99)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1999. to appear.

[PSS98a] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)-automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[PSS98b] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *4th Intl. Conf. TACAS'98*, volume 1384 of *Lect. Notes in Comp. Sci.*, pages 151–166. Springer-Verlag, 1998.

[WZ91] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 4 1991.

[ZPG00] L. Zuck, A. Pnueli, and B. Goldberg. Translation validations of loop optimizations in optimizing compliers. 7 2000.