VOC: A Translation Validator for Optimizing Compilers [1,2]

Lenore Zuck [3] Amir Pnueli [4] Yi Fang [3] Benjamin Goldberg [3]

Abstract

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of safety-critical components of systems. Most formal verification methods verify the correctness of a high-level representation of the system against a given specification. However, if one wishes to infer from such a verification the correctness of the code which runs on the actual target architecture, it is essential to prove that the high-level representation is correctly implemented at the lower level. That is, it is essential to verify the the correctness of the translation from the high-level source-code representation to the object code, a translation which is typically performed by a compiler (or a code generator in case the source is a specification rather than a programming language).

Formally verifying a full-fledged optimizing compiler, as one would verify any other large program, is not feasible due to its size, ongoing evolution and modification, and, possibly, proprietary considerations. The *translation validation* method used in this paper 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.

The paper presents VOC–a methodology for translation validation of optimizing compilers. We distinguish between *structure preserving* optimizations, for which we establish a simulation relation between the source and target code based on computational induction, and *structure modifying* optimizations, for which we develop specialized "meta-rules". The paper also describes VOC-64—a prototype translation validator that automatically produces verification conditions for the global optimizations of the SGI Pro-64 compiler.

[1] This research was supported in part by NSF grant CCR-0098299, ONR grant N00014-99-1-0131, and the Minerva Center for Verification of Reactive Systems, a gift from Intel, a grant from the German - Israel Foundation for Scientific Research and Development.
[2] Email: {zuck,amir,yifang,goldberg}@cs.nyu.edu
[3] Department of Computer Science, New York University Contact
[4] Department of Computer Science, Weizmann Institute of Science

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

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. Verifying the correctness of modern optimizing compilers is challenging due to the complexity and reconfigurability of the target architectures and the sophisticated analysis and optimization algorithms used in the compilers.

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

The introduction of new classes of microprocessor architectures, such as the EPIC class exemplified by the Intel IA-64 architecture, places an even heavier responsibility on optimizing compilers. This is due to the expectation that static compile-time dependence analysis and instruction scheduling could exploit instruction-level parallelism so as to compete favorably with other architectures, such the superscalar class of machines, where dependences are determined and instructions are reordered at run time by the hardware. As a result, a new family of sophisticated optimizations have been developed and incorporated into compilers, such as the Trimaran and the SGI Pro-64 compilers, targeted at EPIC architectures

Prior work ([PSS98a]) developed a tool for translation validation, CVT, which managed to automatically verify translations involving about 10,000 lines of source code in about 10 minutes. The success of this tool 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.

Other approaches [Nec00,RM00] considered translation validation of less restricted languages allowing, for example, nested loops. They also considered a more extensive set of optimizations. However, the methods proposed there were restricted to *structure preserving* optimizations, and could not directly deal with more aggressive optimizations such as *loop distribution* and *loop tiling* which are used in more advanced optimizing compilers.

Our ultimate goal is to develop a methodology for the translation validation of advanced optimizing compilers, with an emphasis on EPIC-targeted compilers, and the aggressive optimizations characteristic to such compilers.

Our methods will handle an extensive set of optimizations and can be used to implement fully automatic certifiers for a wide range of compilers, ensuring an extremely high level of confidence in the compiler in areas, such as safety-critical systems and compilation into silicon, where correctness is of paramount concern.

In this paper we develop the theory of a correct translation and describe voc-64—our prototype translation validator that automatically produces verification conditions for the global optimizations of the SGI Pro-64 compiler.

The theory of correct translation provides a precise definition of the notion of a target program being a correct translation of a source program, and the methods by which such a relation can be formally established. We distinguish between *structure preserving* optimizations which admit a clear mapping of control points in the target program to corresponding control points in the source program. Most high-level optimizations belong to this class. For such transformation, we apply the well known method of *simulation* which relates the execution between two target control points to the corresponding source execution. A more challenging class of optimizations does not guarantee such correspondence and, for this class, we have developed specialized approaches to which we refer as *meta-rules*. Typical optimizations belonging to this class are *loop distribution* and *fusion*, *loop tiling*, and *loop interchange*.

One of the side-products we anticipate from this work is the formulation of a validation-oriented instrumentation, which will instruct the writer of future compilers how to incorporate into the optimization modules appropriate additional outputs which will make validation straightforward. This will lead to a theory of construction of *self-certifying* compilers.

voc-64, our translation validator, produces verification conditions (VCs) for code produced by the SGI Pro-64 compiler, which can later be sent to be verified by cvt [PSS98a]. The verification conditions produced closely follow our theory: voc-64 finds control points, then builds control and data abstractions and program annotations, and, based on these, it builds the VCs.

1.1 Related Work

The work here is an extension of the work in [PSS98a], and we intend to use the tools developed there in our implementation of the validating tool studied here. The work in [Nec00] covers some important aspects of our work. For one, it extends the source programs considered from a single-loop program to C programs with arbitrarily nested loop structure. An additional important feature is that the method requires no compiler instrumentation at all, and applies various heuristics to recover and identify the optimizations performed and the associated refinement mappings. The main limitation apparent in [Nec00] is that, as is implied by the single proof method described in the report, it can only be applied to structure-preserving optimizations. In contrast, our work can also be applied to structure-modifying optimizations, such as the ones associated with aggressive loop optimizations which are a major component of optimizations for modern architectures, though we do not discuss it in this paper.

Another related work is [RM00] which proposes a comparable approach to translation validation, where an important contribution is the ability to handle pointers in the source program. However, the method proposed there assumes *full* instrumentation of the compiler, which is not assumed here or in [Nec00].

More weakly related are the works reported in [Nec97] and [NL98], which do not purport to establish full correctness of a translation but are only interested in certain "safety" properties. However, the techniques of program analysis described there are very relevant to the automatic generation of refinement mappings and auxiliary invariants.

1.2 General Strategy for Translation Validation of Optimizing Compilers

The compiler receives a *source program* written in some high-level language, translates it into an *Intermediate Representation (IR)* and then applies a series of optimizations to the program, starting with classical architecture-independent *global* optimizations, and then architecture-dependent ones, such as register allocation and instruction scheduling. Typically, these optimizations are performed in several passes (up to 15 in some compilers), where each pass applies a certain type of optimization. Translation validation provides a proof for the correctness of each such optimization pass, where a successful validation results in a proof-script confirmation, and an unsuccessful validation results in a counterexample.

The general approach to establishing a correct correspondence between the target and source code is based on *refinement* and is proved by *simulation*. According to this approach, we establish a *refinement mapping* indicating how the relevant source variables correspond to the target variables or expressions. The proof is then broken into a set of *verification conditions* (also called *proof obligations*), each of which claiming that a segment of target execution corresponds to a segment of source execution. In some of the cases, the proof obligations are not valid by themselves, and then it is necessary to introduce *auxiliary invariants* which provably hold at selected points in the program. The proof obligations are then shown to be valid under the assumption of the auxiliary invariants.

In general terms, our strategy is to first give common semantics to the source and target languages using the formalism of *Transition Systems* (TS's). The notion of a target code T being a correct implementation of a source code 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.

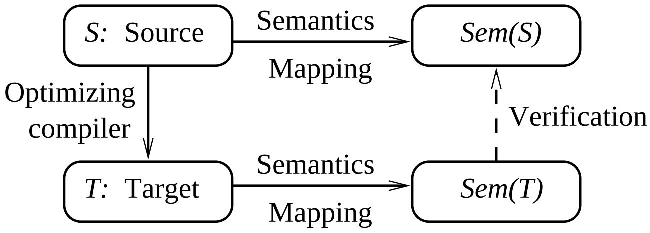

Fig. 1. Refinement Completes the Picture

If only minor optimizations, or no optimizations at all (a debugging mode supported by most compilers), are performed, 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 this simpler case all that is required is to establish the validity of the set of these verification conditions. Under the (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 the 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 validating tool will need additional information which specifies which optimizing transformations have been applied in the current translation. This additional information can either be provided by the compiler or inferred by a set of heuristics and analysis techniques we plan to develop. A major component of our research is the instrumentation of optimizing compilers to provide *program annotations*. These annotations will be processed by the validation tool to form *invariant assertions* at selected control points. The verification conditions will then be augmented by these invariants.

2 The Model

In this section we briefly describe the intermediate code, which we assume to be the input and output language of the optimizing compiler, and the transition system model which our formal model.

2.1 Intermediate Code

The intermediate code is a three-address code. It is described by a *flow graph*, which is a graph representation of the three-address code. Each node in the flow graph represents a *basic block*, that is, a sequence of statements that are always executed in their entirety and contain no branches, and the edges represent the flow of control.

Example 1 Consider the C program in Fig. 2 and its translation, by SGI Pro-64 into intermediate code. The program computes in `r` the summation of the element-wise product of two arrays, the pointers to whose respective initial addresses are `sender` and `weight`. The last element of the array pointed to by `sender` is at `end_sender`.

```
void three(float *r, float *s,
           float *w, float* e)
{
  register float *receiver,
    *weight, *sender, *end_sender;
  receiver = r;
  sender = s;
  weight = w;
  end_sender = e;
  for (; sender <= end_sender; )
       *receiver +=
         (*sender++) * (*weight++)
}
```

```
B0  receiver <- r
    sender  <- s
    weight  <- w
    end_sender <- e
B1  WHILE (sender <= end_sender)
B2  BLOCK
    .t264 <- sender
    sender <- .t264 + 4
    .t265 <- weight
    weight <- .t265 + 4
    [receiver] <- [receiver] +
                  [.t264] * [.t265]
    END_BLOCK
B4  RETURN
```

Fig. 2. A C-Program and its Intermediate Code

The flow graph of the program in Fig. 2, consists of the initial block `B0`, the terminal block `B4`, and the inner blocks `B1`, and `B2`, `B0` leads to `B1`, `B1` leads to `B2` and `B4`, and `B2` leads to `B1`.

For simplicity of exposition, we assume the variables and memory locations do not intersect. At this stage of our work, we ignore aliasing.

Transition Systems

In order to present the formal semantics of source and intermediate code we introduce *transition systems* TS's, a variant of the *transition systems* of [PSS98b]. A *Transition System* $S = \langle V, \mathcal{O}, \Theta, \rho \rangle$ is a state machine consisting of:

- V a set of *state variables*,
- $\mathcal{O} \subseteq V$ a set of *observable variables*,
- Θ an *initial condition* characterizing the initial states of the system, and
- ρ a *transition relation*, 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 $x \in V$, we denote by $s[x]$ the value that s assigns to x. As in our discussion of basic blocks, the transition relation refers to both unprimed and primed versions of the variables, where the primed versions refer to the values of the variables in the successor states, while unprimed versions of variables refer to their value in the pre-transition state. Thus, e.g., the transition relation 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 (pre-transition) state.

The observable variables are the variables we care about. When comparing two systems, we will require that the observable variables in the two systems match. We require that all variables whose values are printed by the program be identified as an observable variables. If desired, we can also include among the observables the history of external procedure calls for a selected set of procedures.

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 transition relation, i.e. $\langle s_i, s_{i+1}\rangle \models \rho$ for every i, $0 \leq i+1 < |\sigma|$ [5].

Example 2 We translate the intermediate code in Fig. 2 into a TS. The set of state variables V includes the observables `r`, `s`, `w`, `e`, and `[r]`, the local variables `receiver`, `sender`, `weight`, `end_sender`, and the temporaries `.t264` and `.t265`. We also include in V the control variable (program counter) π that points to the next statement to be executed. The range of π is $\{\mathtt{B0}, \mathtt{B1}, \mathtt{B2}, \mathtt{B4}\}$. The initial condition, given by $\Theta\colon \pi = \mathtt{B0}$, states that the program starts at location (i.e. block) `B0`. As observables, we take $\mathcal{O} = \{\mathtt{r}, \mathtt{s}, \mathtt{w}, \mathtt{e}\}$.

The transition relation ρ can be presented as the disjunction of four disjuncts $\rho \;=\; \rho_{01} \vee \rho_{12} \vee \rho_{21} \vee \rho_{14}$, where ρ_{ij} describes all possible moves from `Bi` to `Bj` without passing through intermediate blocks.

E.g., ρ_{21} is:

$$(\pi = 1) \;\wedge\; (.\mathtt{t264}' = \mathtt{sender}) \;\wedge\; (\mathtt{sender}' = .\mathtt{t264}' + 4) \;\wedge\; (.\mathtt{t265}' = \mathtt{weight}) \;\wedge$$
$$(\pi' = 2) \;\wedge\; (\mathtt{weight}' = .\mathtt{t265} + 4) \;\wedge\; ([\mathtt{receiver}]' = [\mathtt{receiver}] + [.\mathtt{t264}]' \cdot [.\mathtt{t265}]')$$

When describing a transition relation, we adopt the convention that of mentioning only the variables whose values are changed while leaving the others implicitly intact.

A computation of the program is starts with `B0`, then led (by ρ_{01}) to `B1`, it can then cycle to `B2` (by ρ_{12}) and back to `B1` (by ρ_{21}) several times, then leads (by ρ_{14}) to `B4`, where it terminates. The state reached at each block is described by the values assigned to the variables.

The translation of an intermediate code into a TS is straightforward; we therefore assume that all code we are dealing with here is described by a TS.

Comparison and Refinement between TS*s*

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S\rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T\rangle$ be two TS's, to which we refer as the *source* and *target* TS's, respectively. Such two systems are called *comparable* if there exists a one-to-one correspondence between the observables of P_S and those of P_T. To simplify the notation, we denote by $X \in \mathcal{O}_S$ and $x \in \mathcal{O}_T$ the corresponding observables in the two systems. We say that P_T is

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

a *correct translation* (*refinement*) of P_S if they are comparable and, for every finite (i.e., terminating) P_T-computation $\sigma^T : \sigma_0^T, \ldots, \sigma_m^T$, there exists a finite P_S-computation $\sigma^S : \sigma_0^S, \ldots, \sigma_k^S$, such that $s_m^T[x] = s_k^S[X]$ for every $x \in \mathcal{O}_T$.

Our goal is therefore to provide an automated system that will establish (or refute) that a given target code implements a given source code, where both are expressed as TSs.

3 VOC: Structure-Preserving Transformations

Consider a source TS $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and target TS $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ that are comparable. Let CP_S be a cutpoint set of the source nodes, i.e., a set that includes the initial and terminal blocks (nodes) and at least one node from each loop, and similarly let CP_T be a cutpoint set in the target. For every two nodes i and j in CP_S (resp. CP_T) such that there is a simple path between `Bi` and `Bj` in the source (resp. target), let ρ_{ij}^S (RESP. ρ_{ij}^T) describe the transition relation between blocks `Bi` and `Bj`. Let `pc` denote locations in CP_S, and π denote locations CP_T.

In order to establish that P_T is a correct translation of P_S for the cases that P_T does not alter the structure of P_S in a major way, we introduce, in Fig. 3, the proof rule VALIDATE. The rule is an elaboration of the computational induction approach ([Flo67]) that offers a proof methodology to validate that one program *refines* another. This is achieved by establishing a *control mapping* from target to source locations, a *data abstraction* mapping from source to target variables, and proving that they are preserved with each step of the target program. Some caution is needed in order to guarantee that VALIDATE can handle minor structure changes, for example, as in loop invariant code motion when assignments in the target occur before their counterparts in the source. It is actually a difficult problem to devise such a proof rule – the current version of VALIDATE succeeds a series of proof rules that were sound but failed to handle some common optimizations. This version of VALIDATE has been successful in handling numerous examples, and we believe it is capable of handling all the optimizations that do not involve major structure modifications (as in the various loop optimizations.)

The control abstraction κ in part (1) of VALIDATE is the standard Floyd control mapping. The invariants φ_i in part (2) are program annotations that are expected to be provided by compiler from its data flow analysis. Intuitively, their role is to carry information in between basic blocks. The data abstraction α in part (3) is a variant of the standard Floyd-like data abstraction. The two differences are that we allow for α to be partial, and to include guards (the p_is.) The motivation for allowing α to be partial is to accommodate situations, that occur for example in dead code elimination, where source variables have no correspondence in the target. The motivation for allowing α to contain guards is to accommodate situations, that occur for example in loop invariant code motion, where at some points of the execution, source variables have no

(i) Establish a *control abstraction* $\kappa \colon \mathsf{CP}_T \to \mathsf{CP}_S$ mapping each value of the target control variable `pc` into a corresponding value of the source control variable π. The control abstraction should map the initial and terminal blocks of the target into the initial and terminal locations of the source. Some additional points may have to be added to the CP_S in order to define κ.

(ii) For each basic block `Bi`, form an *invariant* φ_i that may refer only to concrete (target) variables.

(iii) Establish a *data abstraction*

$$\alpha : (p_1 \to v_1 = E_1) \ \wedge\ \cdots\ \wedge\ (p_n \to v_n = E_n)$$

assigning to *some* source state variables $v_i \in V_S - \{\pi\}$ an expression E_i over the target state variables, conditional on the (target) boolean expression p_i. Note that α may contain more than one clause for the same variable.

(iv) For each basic blocks `Bi` and `Bj` such that there is a simple path from `Bi` to `Bj` in the control graph of P_T, form the verification condition

$$C_{ij}: \qquad \varphi_i \ \wedge\ \alpha\ \wedge\ \overset{T}{\rho}_{ij} \quad \to \quad \exists {V_S}' : \overset{S}{\rho}_{\kappa(i),\kappa(j)} \ \wedge\ \alpha' \ \wedge\ \varphi'_j.$$

(v) Establish the validity of all the generated verification conditions.

Fig. 3. The Proof Rule Validate

correspondence in the target, while at other points they do. The guards, thus, describe the conditions under which the source variables can be defined using target variables.

The verification conditions assert that at each (target) transition from `Bi` to `Bj` [6], if the assertion φ_i and the data abstraction hold before the transition, and the transition takes place, then after the transition there exist new source variables that reflect the corresponding transition in the source, while the data abstraction and the assertion φ_j hold in the new state. Hence, 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 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. Since the assertion mentions only target variables, their validity should depend solely on the target code. In most cases, the primed source variables can be easily realized from the code, and

[6] Recall that we assume path described by the transition is simple.

the existential quantification in verification condition (iv) can be eliminated, since the implication $q \rightarrow \exists x' : (x' = E) \wedge r$ is validity-equivalent to the implication $q \wedge (x' = E) \rightarrow r$. However, this may not always be the case and we are forced to leave the existential quantifier in (4).

In [PZP00] we proved a theorem stating that a variant of the proof rule VALIDATE is sound. The proof holds also for this version of the rule, and we omit it here.

In the next section we describe VOC-64 – a tool, currently being developed at NYU, to automatically generate verification conditions (VCs) for programs being compiled by the SGI Pro-64 compiler. Verification of the VCs that are generated by VOC-64 can be performed by CVT, that was developed for the *Sacres* project [PRSS99]. The parts that cannot be handled by CVT can be handled by other theorem provers. Currently, we have been using STeP [MAB+94], and are exploring other packages that can provide similar capabilities.

Before we describe VOC-64 and its capabilities, we give here a example of an application of VALIDATE as generated by VOC. For readability, the output here is "human-aided", but is essentially as produced by the tool. In the next section, we describe the tool in more detail.

3.1 An Application of VALIDATE

This example is derived using VOC-64 which is described in Section 4. Here we sketch some of the sample output. Consider the program of Fig. 2 after a series of optimizations: Copy propagation, dead code elimination, control flow graph optimization (loop inversion), and register variable identification. A simplified version of the resulting code, where we renamed some variables, is in Fig. 4. The annotation (φ_2, denoted `phi2`) is supplied by the compiler (see Section 4).

```
B0  sender <- s
    weight <- w
    IF !(s <= e) GOTO B4
B1  .t266 <- [r]
    .t267 <- e + 4
B2  {phi2: (.t267 = e + 4) &&
     (weight = sender - s + w)
     && (.t266 = [r])}
     .t271 <- sender
     sender <- .t271 + 4
     .t270 <- weight
     weight <- .t270 + 4
     .t266 <- .t266 + [.t271] * [.t270]
     [r] <- .t266
     IF (sender != .t267) GOTO B2
B3   GOTO B5
B4   .t266 <- [r]
B6   RETURN
```

Fig. 4. Annotated Optimized Program

To validate the program, we use the control mapping $\kappa = \{0 \mapsto 0, 2 \mapsto 2, 6 \mapsto 4\}$, and the data abstraction :

$$\alpha : \begin{pmatrix} (\mathtt{SENDER} = \mathtt{sender}) \wedge (\mathtt{RECEIVER} = \mathtt{r}) \wedge (\mathtt{WEIGHT} = \mathtt{weight}) \wedge (\mathtt{S} = \mathtt{s}) \\ \wedge (\mathtt{END\_SENDER} = \mathtt{e}) \wedge (\mathtt{R} = \mathtt{r}) \wedge (\mathtt{W} = \mathtt{w}) \wedge (\mathtt{E} = \mathtt{e}) \wedge ([\mathtt{RECEIVER}] = [\mathtt{r}]) \end{pmatrix}$$

where upper case latter denote source (abstract) variables, and lower case letter denote their target (concrete) counterparts.

The VC C_{22} we obtain for the transition from B2 to B2, after simplification (including the removal of the existential quantifier) is:

$$C_{22}\colon \ : \varphi_2 \ \wedge \ \overset{T}{\rho}_{22} \ \wedge \ \overset{S}{\rho}_{22} \ \rightarrow \ \alpha' \ \wedge \ \varphi_2'$$

where $\overset{T}{\rho}_{22}$ is define by:

$$\begin{pmatrix} (\texttt{pc} = 2) \ \wedge \ (.t266' = .t266 + [\texttt{sender}] * [\texttt{weight}]) \ \wedge \ (.\texttt{t270}' = \texttt{weight}) \\ \wedge \ (.t271' = \texttt{sender}) \ \wedge \ (\texttt{sender}' = \texttt{sender} + 4) \ \wedge \ (\texttt{weight} = \texttt{weight} + 4) \\ \wedge \ ([r]' = .t266 + [\texttt{sender}] * [\texttt{sender}]) \ \wedge \ (\texttt{sender} + 4 \neq .\texttt{t267}) \ \wedge \ (\texttt{pc}' = 2) \end{pmatrix}$$

and $\overset{S}{\rho}_{22}$ is defined by:

$$\begin{pmatrix} (\pi = 2) \ \wedge \ (\pi' = 2) \ \wedge \ ([\texttt{RECIEVER}]' = [\texttt{RECEIVER}] + [\texttt{SENDER}] * [\texttt{WEIGHT}]) \ \wedge \\ (\texttt{WEIGHT}' = \texttt{WEIGHT} + 4) \ \wedge \ (\texttt{SENDER} + 4 \leq \texttt{END\_SENDER}) \ \wedge \ (\texttt{SENDER}' = \texttt{SENDER} + 4) \end{pmatrix}$$

We also have:

$$\alpha'\colon \begin{pmatrix} (\texttt{SENDER}' = \texttt{sender}') \ \wedge \ (\texttt{RECEIVER}' = \texttt{r}') \ \wedge \ (\texttt{WEIGHT}' = \texttt{weight}') \\ \wedge \ (\texttt{END\_SENDER}' = \texttt{e}') \ \wedge \ (\texttt{S}' = \texttt{s}') \ \wedge \ (\texttt{R}' = \texttt{r}') \ \wedge \ (\texttt{W}' = \texttt{w}') \ \wedge \ (\texttt{E}' = \texttt{e}') \end{pmatrix}$$

and $\varphi_2'\colon (.t267' = \texttt{e}' + 4) \ \wedge \ (\texttt{weight}' = \texttt{sender}' - \texttt{s}' + \texttt{w}') \ \wedge \ (.\texttt{t266}' = [\texttt{r}']')$

The other verification conditions are constructed similarly. They are all trivial to verify.

4 VOC-64: A translation Validator for SGI Pro-64

In this section we give an overview of voc-64 and show some sample output as produced by voc-64 for the source program of Fig. 2 and its target of Fig. 4. The code, a (draft) manual, and examples can be found in *http://www.cs.nyu.edu/validation/tvoc*.

4.1 Description of voc-64 *'s Components*

Parser (*wh.cxx* and *wh.h*). The intermediate language of SGI Pro-64 (or SGI for short) is WHIRL. After each round of optimization, the compiler outputs ASCII formatted WHIRL code, which can be read by a parser and translated back into a graphic representation.

Cutpoint Set(*ts_more.cxx*). voc-64 computes cutpoint sets, CP_SET, for both source and target as follows. The cutpoint sets include initial and terminal blocks. Whenever a loop is encountered, if there exists a block in the loop that either contains an assignment statement, or has only outgoing edges that lead to blocks that are internal to the loop, then the first such block is included in the cutpoint set. If no such block exists, then the first block of the loop is included in the cutpoint set. Both initial and terminal blocks are also included in the cutpoint set.

For the example of the previous section, voc-64 computes the cutpoint set $\{0, 2, 4\}$ for the source, and $\{0, 2, 6\}$ for the target.

Paths (*ts_more.cxx*). voc-64 computes sets `CP_PATHS` of paths for each source and target. The set includes all *simple* paths (and cycles) in between two points in the appropriate `CP_SET`, i.e., paths that do not contain as intermediate points any other point in `CP_SET`.

For our example, voc-64 computes for the source is
$\texttt{CP\_PATHS} = \{\texttt{B0} \to \texttt{B1} \to \texttt{B2}, \texttt{B2} \to \texttt{B1} \to \texttt{B2}, \texttt{B0} \to \texttt{B1} \to \texttt{B4}, \texttt{B2} \to \texttt{B1} \to \texttt{B4}\}$

Transition Relation For each path $\texttt{Bi} \to \texttt{Bj}$ in `CP_PATHS`, voc-64 computes the τ_{ij}, which is ρ_{ij} without the control information (that is implicit.) The output is `Ps(i,j)` for source, and `Pt(i,j)` for target, that consists of a set of equalities over the appropriate variables and the branch conditions. Thus, ρ^x_{ij} is a conjunct of the terms in `Px(i,j)`, together with $\pi = i \ \wedge \ \pi' = j$. E.g., for `Ps(0,2)` voc-64 produces:

```
(RECEIVER' = R) (SENDER' = S) (WEIGHT' = W) (END_SENDER' = E) (S <= E)
```

Invariants (*ts.cxx*). The set of invariants φ_i voc-64 computes for every basic block `Bi` consists of two types of invariants, those whose source are reachable definitions, and those whose source is loop induction variables. The former help voc-64 deal with optimizations such as copy propagation and code motion, and the latter help voc-64 deal with optimizations such as strength reduction.
Computing Invariants of the 1^{st} *type.* A y-definition d of the form $y := f(x_1, \ldots, x_n)$ is *available on the path from* `Bj` *to* `Bi` if d is the last definition of y in `Bj`, and every path from `Bj` to `Bi` contains no (re)definition of $y, x_1, \ldots, x_n$. An assertion of the form $y = f(x_1, \ldots, x_n)$ is an *invariant at* `Bi` if all the available y-definitions at `Bi` are of the form $y := f(x_1, \ldots, x_n)$. For each point in the cutpoint set, voc-64 computes the set of invariants. Detecting such invariants is similar to computing fixpoints when computing reaching definitions.
Computing Invariants of the 2^{nd} *type*: For every loop induction variables i and j whose initial value i_0 and j_0 respectively, and that get incremented in each iterations by c_1 and j by c_2 respectively, the invariant $j = c_2/c_1 * i + (j_0 - i_0 * c_2/c_1)$ is computed.

Data Abstraction (*ts_more.cxx*). The compiler provides us with the data mapping of each target variable that has a counterpart in the source. For memory access, $a[e]$, is interpreted as $[\&a + \mathit{size} * e]$ and $[E_i] = [e_j]$ holds for any $E_i = e_j$. For a non-parameter source variable V, $V = v$ holds only after both V and v are initialized. That is, if a definition of v occurs in the target code, and no definition of V occurs in the corresponding blocks in the source, we conclude that hoisting of loop invariant code has occurred and that $V = v$

does not hold at that point. For parameters, we take $V = v$ as a precondition.

Thus, in our example, we obtain in the data abstraction both ($\texttt{pc} \in \{0,2,6\}) \rightarrow ([\texttt{R}] = [\texttt{r}])$ and $(\texttt{pc} \in \{2,6\}) \rightarrow ([\texttt{SENDER}] = [\texttt{sender}])$.

Control Mapping (*ts_more.cxx*) voc-64 maps each initial and terminal locations of the target to initial and terminal locations of the source. The mapping of target loops to source loops is accomplished using the information provided by the compiler, and each cutpoint of a target loop is mapped to the cutpoint of the corresponding source loop.

For our example, voc-64 produces {`0 -> 0, 2-> 2, 6 -> 4`}.

Generation of Verification Conditions The construction of the VCs is usually straightforward. An interesting case is, however, when there are multiple paths connecting two cutpoints in both source and target. Assume that the cutpoint set includes `Bi` and `Bj`. Assume further that there are m different paths from `Bi` to `Bj`, each contributing a disjunct P_k to $\rho^T_{i,j}$, and that there are n different paths from $\kappa(\texttt{Bi})$ to $\kappa(\texttt{Bj})$, each contributing a disjunct Q_ℓ to $\rho^S_{\kappa(i),\kappa(j)}$. Then voc-64 generates m VCs C^k_{ij}, $k = 1,\ldots,m$, each of the form $\varphi_i \ \wedge \ \alpha \ \wedge \ P_k \quad \rightarrow \quad \exists {V_S}' \colon \bigvee_{\ell=1}^n Q_\ell \ \wedge \ \alpha' \ \wedge \ \varphi'_j$.

5 A Note on Loop Unrolling

A schematic loop unrolling is in Fig. 5 (where we assume $n \geq c > 1$.) There

`L1: i=1`
`L2: B(i); i=i+1;`
`    if (i<=n) goto L2`
`L3:`
	$\Longrightarrow$	`L1: i=1`
		`L2: B(i); B(i+1); ... ; B(i+c-1);`
		`    i=i+c; if (i+c-1 <= n) goto L2`
		`L3: if (i > n) goto L5`
		`L4: B(i); i=i+1; if (i<=n) goto L4`
		`L5:`

Fig. 5. Loop Unrolling

are several strategies for dealing with loop unrolling. One is to design a meta-rule (see next section) that deals with it directly. Another is to consider loop unrolling as a special case of tiling an $n \times 1$ array with tiles of size c, and then unrolling the innermost loop. A third approach, which we pursue here, is to consider loop unrolling as a structure-preserving transformation and apply Validate to it.

Hence, for simple path between two cutpoints in the target, voc finds its corresponding excution in source and check whether the branch conditions of the two paths are congruent. We demonstrate the approach on the C-code and its translation described in Fig. 6.

C Code

```
long a[100];
void unroll(int n)
{
  int i;
  for(i=0; i<n; i++)
      a[i] = i;
}
```

Source Code (IR)

```
B0  i <- 0
B1  WHILE (i < n)
B2      [&a + i * 8] <- i
        i <- i + 1
B3  RETURN
```

Target

```
B0  IF !(0 < n) GOTO B6
B1  i <- 0
    t1 <- &a
    t2 <- n MOD 3
    IF (t2 = 0) GOTO B4
B2  [t1] <- i
    t1 <- t1 + 8
    i <- i + 1
    IF (t2 = 1) GOTO B4
B3  [t1] <- i
    t1 <- t1 + 8
    i <- i + 1
B4  IF !(i < n) GOTO B6
B5  t10 <- t1
    t11 <- t1 + 8
    t12 <- t1 + 16
    t20 <- i
    t21 <- i + 1
    t22 <- i + 2
    [t10] <- t20
    [t11] <- t21
    [t12] <- t22
    t1 <- t1 + 24
    i <- i + 3
    IF (i < n)
        GOTO B5
B6  RETURN
```

Fig. 6. A Loop Unrolling Example

voc-64 generated the control mapping $\{0 \mapsto 0, 5 \mapsto 2, 6 \mapsto 3\}$, the data abstraction $N = n \ \wedge \ (\mathtt{pc} > 1 \rightarrow (I = i))$ and the invariants

$$
\begin{aligned}
\varphi_0 &: \ \mathrm{T} \\
\varphi_5 &: \ (t1 = 8 \cdot i + \&a) \ \wedge \ (i < n) \ \wedge \ ((n - i) \bmod 3 = 0) \\
\varphi_6 &: \ (t1 = 8 \cdot i + \&a) \ \wedge \ ((n - i) \bmod 3 = 0)
\end{aligned}
$$

For example, for $C_{56} = \varphi_5 \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho}_{56} \rightarrow \exists V'_{S} : \overset{S}{\rho}_{23} \ \wedge \ \alpha' \ \wedge \ \varphi'_6$, where $\overset{S}{\rho}_{23}$ is as per the program of the r-h-s of Fig. 5. voc-64 generated for $\overset{T}{\rho}_{56}$:

$$
\left(\begin{array}{l} (\mathtt{pc} = 5) \ \wedge \ (\mathtt{pc}' = 6) \ \wedge \ i' = i + 3) \ \wedge \ (t1' = t1 + 24) \ \wedge \ ([t1]' = i) \\ \ \wedge \ ([t1 + 8]' = i + 1) \ \wedge \ ([t1 + 16]' = i + 2) \ \wedge \ (i + 3 \geq n) \end{array} \right)
$$

and for $\overset{S}{\rho}_{23}$:

$$
\left(\begin{array}{l} [8 \cdot I + \&A]' = I) \ \wedge \ ([8 \cdot (I + 2) + \&A]' = I + 2) \ \wedge \ ([8 \cdot (I + 1) + \&A]' = I + 1) \ \wedge \\ (\pi = 2) \ \wedge \ (I' = I + 3) \ \wedge \ (I + 1 < N) \ \wedge \ (I + 2 < N) \ \wedge \ (I + 3 \geq N) \ \wedge \ (\pi' = 3) \end{array} \right)
$$

In [PZL01] we give an example of a compilation of the Trimaran compiler that involves loop unrolling, and verify it using Validate. In fact, in one case the validation failed, showing that Trimaran generated target code that could possibly cause a segmentation fault, although the source code would not have.

6 Validating Structure-Modifying (Loop) Optimizations

Since the simulation proof method assumes that the source and target have similar structures, rule Validate cannot be used to validate many of the loop optimizations. Therefore, we propose an alternate methodology for validating loop optimizations, using a set of "meta-rules", each dealing with a *set* of loop optimizations. The soundness of the meta-rules is established separately. Usually, structure-modifying optimizations are applied to small localized sections of the source program, while the rest of the program is only optimized by structure-preserving transformations. Therefore, the general validation of a translation will combine these two techniques.

The first meta-rule covers a wide range of optimizations. In fact, it covers all optimizations in which the loop body itself is not altered (except for substitution of indices.) The obvious cases that are not covered by this meta-rule are loop distribution and fusion, that are covered by a different meta-rule.

This meta rule covers all cases where the only changes the optimization imposes on the loop body are those caused by substitution of control variables. Thus, it covers all unimodular loop transformations, as well as tiling and combinations of the above.

A loop transformation has the general form of Figure 7.

For $(\mathtt{i}_1, \ldots, \mathtt{i}_{\mathtt{k}_1}) \in P_1$ by $\prec_1$ do B1$(\mathtt{i}_1, \ldots, \mathtt{i}_{\mathtt{k}_1})$ $\mathtt{L_S}$: **Source Loop**	For $(\mathtt{j}_1, \ldots, \mathtt{j}_{\mathtt{k}_2}) \in P_2$ by $\prec_2$ do B2$(\mathtt{j}_1, \ldots, \mathtt{j}_{\mathtt{k}_2})$ $\mathtt{L_T}$:**Target Loop**

Fig. 7. A General Loop Transformation

In this representation, we assume that each of the loop bodies, B1 and B2, have some occurrences of the variable $\boldsymbol{i} = (i_1, \ldots, i_{k_1})$ and $\boldsymbol{j} = (j_1, \ldots, j_{k_2})$ respectively; these variables are not modified in either B1 or B2. We use the notation B$(\boldsymbol{k})$ to indicate an execution of block B where the loop control variables $\boldsymbol{i}$ have the value $\boldsymbol{k}$ upon entrance to B.

Our main meta rule, which applies to all transformations in the form of Figure 7, is in Figure 8.

For some $k_1 \times k_2$ *transformation matrix* $\mathbf{T}$ and vector $\boldsymbol{b} = (b_1, \ldots, b_{k_1})$,

(i) $P_1 = \mathbf{T} \cdot P_2 + \boldsymbol{b}$;

(ii) For every $\boldsymbol{j} \in P_2$, $\mathtt{B2}(\boldsymbol{j}) \approx \mathtt{B1}(\mathbf{T} \cdot \boldsymbol{j} + \boldsymbol{b})$;

(iii) For every $\boldsymbol{j_1}, \boldsymbol{j_2} \in P_2$ such that $\boldsymbol{j_1} \prec_2 \boldsymbol{j_2} \in P_2$ and $\mathbf{T} \cdot \boldsymbol{j_2} \prec_1 \mathbf{T} \cdot \boldsymbol{j_1}$,
$\mathtt{B1}(\mathbf{T} \cdot \boldsymbol{j_1} + \boldsymbol{b}); \mathtt{B1}(\mathbf{T} \cdot \boldsymbol{j_2} + \boldsymbol{b}) \approx \mathtt{B1}(\mathbf{T} \cdot \boldsymbol{j_2} + \boldsymbol{b}); \mathtt{B1}(\mathbf{T} \cdot \boldsymbol{j_1} + \boldsymbol{b})$

$$\overline{\mathtt{For}\ \boldsymbol{i} \in P_1\ \mathtt{by} \prec_1 \mathtt{do}\ \ \mathtt{B1}(\boldsymbol{i}) \ \approx\ \mathtt{For}\ \boldsymbol{j} \in P_2\ \mathtt{by} \prec_2 \mathtt{do}\ \ \mathtt{B2}(\boldsymbol{j})}$$

Fig. 8. Meta Proof Rule for Transformation of Figure 7

Condition (1) of the rule states that there is a transformation such that (the polyhedron) P_1 is the result of applying the transformation to P_2, thus, the two loops are defined over the same vectors. Condition (2) of the rule states the loop body of each iteration of the source is obtained by the loop body of the transformed iteration of the target, thus, the loop body of two corresponding iterations is the same. Condition (3) guarantees that any two iterations that are executed in different order in the source and target loops do not depend on one another.

In [ZPG00] we show examples of application of the rule for simple loop interchange, simple loop skewing, and nested (depth=2) loop tiling, loop fusion and loop distribution.

Validation using the proof rule in Figure 8 involves three steps:

(i) Generating P_1, P_2, $\mathbf{T}$, and $\boldsymbol{b}$ and proving that $P_1 = \mathbf{T} \cdot P_2 + \boldsymbol{b}$: The polyhedra P_1 and P_2 can be automatically obtained from the syntactic form of the iterations at hand; the transformation matrix $\mathbf{T}$ and the vector $\boldsymbol{b}$ can be obtained by proper instrumentation of the compiler; it is also conceivable that they can be obtained by simple heuristic method, especially when only a single transformation is applied at a time. The equivalence of P_1 and $\mathbf{T} \cdot P_2 + \boldsymbol{b}$ can be checked by any suitable mathematic package, e.g., Mathematica [Wol99].

(ii) Checking that $B_1(\mathbf{T}\cdot\boldsymbol{j}+\boldsymbol{b}) \approx B_2(\boldsymbol{j})$ for every $\boldsymbol{j} \in P_2$: Often, once $\mathbf{T}$ and $\boldsymbol{b}$ are known, this amounts to merely checking that $B_1(\mathbf{T}\cdot\boldsymbol{j}+\boldsymbol{b}) = B_2(\boldsymbol{j})$. When the loop bodies are significantly altered, the equivalence may be established by CVT-like tools.

(iii) Validating that any two iterations whose relative order in the source and the destination are different do not depend on one another: The first step is to find the "potentially offensive" pairs of iterations. This can be done by using the same mathematical package used in step (1). For each of these pairs (sets of them may be represented symbolically), the equivalence of the two bodies can be established using the decision procedure underlying CVT.

7 Conclusion

This paper presented VOC, the theoretical framework for translation validation of optimizing compilers. We described VOC-64, a tool we are developing to perform automatic translation validation for the SGI Pro-64 compiler, and gave examples of the VCs it generates. VOC-64 currently generates verification conditions for all the classical global optimizations of the SGI Pro-64, as well as for loop unrolling.

We are currently working on extending VOC to handle other loop optimizations. One of the main problems facing us is the current need to interface with a "heavy" mathematical package, which we hope to overcome shortly. We are also working on the interface with CVT to generate the formal proofs of the VCs produced by VOC.

In the near future we are planning to embark on developing the theory for dealing with instruction scheduling and other machine-dependent optimizations.

References

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

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

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

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

[Nec97] G.C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. Princ. of Prog. Lang.*, pages 106–119, 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.

[NL98] G.C. Necula and P. Lee. The design and implementation of a certifying compilers. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 1998*, pages 333–344, 1998.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In *In N. Halbwachs and D. Peled, editors, Proc.* 11^{st} *Intl. Conference on Computer Aided Verification (CAV'99), volume 1633 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 455–469, 1999.

[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 *In B. Steffen, editor, Proc.* 4^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), volume 1384 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 151–166, 1998.

[PZP00] A. Pnueli, L. Zuck, and P. Pandya. Translation validation of optimizing compilers by computational induction. Technical report, Courant Institute of Mathematical Sciences, New York University, 2000.

[PZL01] A. Pnueli, L. Zuck, and R. Leviathan. Validation of Optimizing Compilers. Technical report, Computer Science Department, NYU (CIMS/CS/NYU.) Available in `http://www.cs.nyu.edu/validator/pubs.html`

[RM00] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, Trento, July 2000.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, Comp. Sci. Laboratory, SRI International, Menlo Park, CA, 1993.

[Wol99] S. Wolfram. *The Mathematica Book*. Cambridge University Press, 1999.

[ZC91] H. Zima and B. Chapman. *Supercompiler for Parallel and Vector Computers*. Addison-Wesley. 1991.

[ZPG00] L. Zuck, A. Pnueli, and B. Goldberg. Translation Validation of Loop Optimizations in Optimizing Compilers. Technical report, CIMS/CS/NYU. Available in `http://www.cs.nyu.edu/validator/pubs.html`