Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 21–35
www.elsevier.com/locate/entcs

Improved Invariant Generation for TVOC

Yi Fang[1]

Microsoft Corp.
Redmond WA, U.S.A

Lenore D. Zuck [2,3]

Computer Science Department
University of Illinois at Chicago
Chicago MI, U.S.A

Abstract

The NYU TVOC project applies the method of translation validation to verify that optimized code is semantically equivalent to the unoptimized code, by establishing, for each run of the optimizing compiler, a set of verification conditions (VCs) whose validity implies the correctness of the optimized run. The core of TVOC is TVOC-SP, that handles structure preserving optimizations, i.e., optimizations that do not alter the inner loop structures. The underlying proof rule, **Val**, on whose soundness TVOC-SP is based, requires, among other things, to generating invariants at each "cutpoint" of the control graph of both source and target codes. The current implementation of TVOC-SP employs somewhat naïve fix-point computations to obtain the invariants. In this paper, we propose an alternative method to compute invartiants which is based on simple data-flow analysis techniques.

Keywords: Translation validation, invariant generation, data abstraction, data-flow analysis.

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 focus on verification of specifications with respect to requirements, and high-level code with respect to specifications. However, if one is to prove that the high-level specification is correctly implemented in low-level code, one needs to verify the compiler which performs the translations. Verifying the correctness

[1] Email: yfang@microsoft.com
[2] Email: lenore@cs.uic.edu
[3] This research was supported in part by NSF grant CCR-0205571 and SRC grant 2004-TJ-1256.

doi:10.1016/j.entcs.2006.06.016

of modern optimizing compilers is challenging because of the complexity and reconfigurability of the target architectures, as well as 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. The *Translation Validation* approach, which was introduced in [9], offers an alternative to the verification of translators in general and of compilers in particular: Rather than verify 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.

In the past five years we have been working towards developing a methodology for fully automatic translation validation of optimizing compilers (see, e.g., [14,15,3]). The methodology consists of a theory of correct translation, and a tool suite, TVOC, that performs translation validation for Intel's ORC compiler. The theory distinguishes between *structure preserving* optimizations, that admit a clear mapping of control and data values in the target program to corresponding control and data values in the source program, and *structure modifying* optimizations that admit no such clear mapping. Most high-level optimizations are structure preserving. TVOC consists of two main parts: TVOC-SP that handles the structure preserving transformations, and TVOC-LOOP that handles loop reordering transformations, which are reduced to equivalence checking handled by TVOC-SP.

Given source (pre-optimization) and target (post-optimization) codes, TVOC-SP constructs verification conditions (VCs) whose validity implies the semantic equivalence between the two, and passes the VCs to a theorem prover. The theory underlying TVOC-SP is the Floyd-style proof rule **Val**. The proof rule **Val** differs from similar proof rules by requiring, in addition to the usual data and control mappings, the construction, at selected control points, of *invariants* that carry information among basic blocks. The strength of the invariants generated determines the power of the tool.

In this paper we study the invariant generation that **Val** calls for, and propose a methodology to obtain stronger invariants than obtained in the current implementation of TVOC-SP. Currently (see [3]), TVOC-SP performs rather naïve fixed-point computations to obtain the data mappings and the invariants. The gist of our idea is to translate the input to TVOC, which is textual Whirl ([12]), into Static Single Assignment (SSA) form, and perform *simple* data-analysis to obtain both the data mapping and the invariants required to apply **Val**. We also present a version of **Val** that is more suitable for our needs. The techniques described here were implemented (see [7]) and produce superior results to those of TVOC in almost all cases.

The paper is organized as follows: In Section 2 we present the new version of rule **Val** and discuss how it differs from its predecessors. In Section 3, which is the heart of paper, we describe the new method of invariant generation. In Section 4 we describe a new computation of data mapping that the new invariants generation entails. We conclude in Section 5.

Related Work

The work in [9] developed a tool for translation validation, CVT, that succeeded in automatically verifying translations involving approximately 10,000 lines of source code in about 10 minutes. The success of CVT critically depends on some simplifying assumptions that restrict the source and target to programs with a single external loop, and assume a very limited set of optimizations.

Other approaches [8,11] considered translation validation of less restricted languages than considered in [9], 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 that involve code motion or loop reordering transformations.

The theory of correct translation validation on which the current paper is based appears in, e.g., [14,15], and [3] presents the TVOC tool suite.

2 The General Framework

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.

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 is executed in its entirety and contains no branches. The edges of the graph represent the flow of control.

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 [10]. 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. 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, where we treat each I/O device as a variable, and each I/O operation removes/appends elements to the corresponding variable. If desired, we can also include among the observables the history of external procedure calls for a selected set of procedures. When comparing two systems, we will require that the observable variables in the two systems match.

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 such that every two consecutive states are related by the transition relation. I.e., $s_0 \models \Theta$ and $\langle s_i, s_{i+1} \rangle \models \rho$ for every i, $0 \leq i + 1 < |\sigma|$ [4] .

A transition system S is called *deterministic* if the observable part of the initial condition uniquely determines the rest of the computation. That is, if S has two computations $s_0, s_1, \ldots$ and $t_0, t_1, \ldots$ such that the observable part (values of the observable variables) of s_0 agrees with the observable part of t_0, then the two computations are identical. We restrict our attention to deterministic transition systems and the programs which generate such systems. Thus, to simplify the presentation, we do not consider here programs whose behavior may depend on additional inputs which the program reads throughout the computation. It is straightforward to extend the theory and methods to such intermediate input-driven programs.

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. A source state s is defined to be *compatible* with the target state t, if s and t agree on their observable parts. That is, $s[X] = t[x]$ for every $x \in \mathcal{O}_T$. We say that P_T is a *correct translation* (*refinement*) of P_S if they are comparable and, for every $\sigma_T : t_0, t_1, \ldots$ a computation of P_T and every $\sigma_S : s_0, s_1, \ldots$ a computation of P_S such that s_0 is compatible with t_0, σ_T is terminating (finite) iff σ_S is and, in the case of termination, their final states are compatible.

Our goal is to provide an automated method that will establish (or refute) that a given target code correctly implements a given source code, where both are expressed as Ts's.

Assume P_S and P_T are *source* and *target* programs given in some intermediate language, that we wish to show are equivalent. Furthermore, we assume that both P_S and P_T are in SSA-form ([6]), which allow for more powerful data mappings and invariant generation than those allowed by the non-SSA-formed programs ([5,4] [5]).

The translation of the codes into Ts's is straightforward. We therefore assume that both programs are Ts's. Hence we refer to both IR programs and their Ts's by the same name. For each of the programs, we compute a *cutpoint* set, i.e., a set of control points that includes the entry point, exit point, and at least one control point of each loop. The source cutpoint set is denoted by CP_S, and the target's is denoted by CP_T. A *simple* path is a path between control cutpoints that has no

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

[5] See also ipf-orc.sourceforge.net/ORC-documentation.htm.

(i) Establish a *control abstraction* κ: $\mathsf{CP}_T \to \mathsf{CP}_S$ that maps initial and terminal points of the target into the initial and terminal points of the source.

(ii) For each target cut point i in CP_T, form an *target invariant* $\varphi_T(i)$ that may refer only to target variables.

(iii) For each source cut point i in CP_S, form a *source invariant* $\varphi_S(i)$ that may refer only to source variables.

(iv) Establish, for each target cut point i in CP_T, a *data abstraction*

$$\alpha(i) : (v_1 = E_1) \ \wedge \ \cdots \ \wedge \ (v_n = E_n)$$

assigning to *some* non-control source state variables $v_k \in V_S$ an expression E_k over the target state variables.

Note that $\alpha(i)$ is allowed to be partial, i.e., it may contain no clause for some variables. It is required that, for every observable variable (whose target counterpart is v) and every terminal point t, $\alpha(t)$ has a clause $V = E(V)$ where $E(V)$ is an expression over the target observable variables.

(v) For each cut points i and j such that there is a simple path from i to j in the control graph of P_T, form the verification condition

$$C_{ij}: \quad \left(\varphi_T(i) \ \wedge \ \varphi_S(\kappa(i)) \ \wedge \ \alpha(i) \ \wedge \ \overset{T}{\rho}_{ij} \quad \rightarrow \quad \exists V_S' : \overset{S}{\rho}_{\kappa(i),\kappa(j)} \ \wedge \ \alpha'(j) \ \wedge \ \varphi'_S(\kappa(j)) \ \wedge \ \varphi'_T(j). \right)$$

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

Fig. 1. The Proof Rule **Val**

intermediate cutpoints.

The proof rule **Val**, presented in Fig. 1, describes how to establish that P_T is a correct translation of P_S. The rule calls for computing a *control mapping* from target cutpoints into source cutpoints, and a *data mapping* that maps (some) source variables into expressions over the target variables. In addition, *invariants* are computed at each control points of both P_S and P_T. The invariants allows to carry information between basic blocks and give **Val** extra edge, missing from other validation techniques. The proof rule presented here is a variant of some predecessors (see, e.g., [15]).

In Fig. 1, upper cases letters are used to denote variables in the P_S, and lower case letters are used to denote variables in P_T.

For each cutpoint i and cutpoint j, ρ_{ij} describes the generalized transition relation between i and j, i.e., an assertion of the type $\tau(V, V')$ where V is the variables before the transition and V' is the variables after the transition. For example, consider the example on the left hand-side of Fig. 2. There, for the generalized transition from location 1 to location 2 we have: ρ_{12}: $(\mathtt{PC} = 1) \ \wedge \ (\mathtt{PC}' = 2) \ \wedge \ ((B_1 = 1 \ \wedge \ N_1' = 500) \vee (B_1 \neq 1 \ \wedge \ N_2' = 0)) \ \wedge \ \mathit{pres}(V_S - \{N_1, N_2, \mathtt{PC}\})$. The first two conjuncts refer to the value of the program counter, that is 1 before, and 2 after, the transition. The second conjunct describes the effects of the two paths that can occur. The third conjunct specifies the transition preserves the values of all source variables but for PC, N_1 and N_2.

The invariants $\varphi(i)$ are "program annotations" that are computed by the translation validator. Their role is to carry information in between cutpoints. Their invariance is proved in step (5).

The main differences between this version of **Val** and previous versions of it are:

(i) In previous versions of **Val**, only target invariants are mentioned. However, in the tool TVOC, both source and target invariants are used. From a theoretical point of view, nothing is gained by adding source invariants since target invariants together with the data mapping α can capture the source invariants. From a practical point of view, however, it is simpler to compute the source invariants directly, which is explicit here.

(ii) The data abstraction α in previous versions of the rule includes guards that refer to target variables. Here we choose a simpler form that does not include the guards, but that establishes a separate data mapping for each target cutpoint (to accommodate code motion), each may refer only to a subset of the source variables. The form here is weaker, since it (implicitly) allows only for target control variable to be the guard, and not any predicate over target variables. However, the implementation of TVOC computes only data abstraction with the the target control variable as the guard, thus we choose here a version that corresponds to the tool.

(iii) The existential quantification in the right-hand-side of the verification condition C_{ij} is discussed in [14]. As pointed out in [15], this existential conjunct can be eliminated by including, in the left-hand-side of the implication, the conjunct

$$\bigvee_{\pi \in \{\text{simple paths from } \kappa(i)\}} \rho_\pi^S$$

where ρ_π^S is the transition relation corresponding to the path π, thus replacing the existential quantifier with a finite (and small) conjunction.

2.1 Applying **Val***: An Example*

In order to apply **Val**, the translation validation tool should compute the cutpoint sets, control mapping, simple paths, generalized transition relations (for the simple paths), invariants, and data abstraction. Armed with the above, the tool can then construct the VCs (in step (5) of **Val**) and send them to a theorem prover. We keep the computation of the cutpoint sets, control abstraction, simple paths, as in the current TVOC-SP (see [14,3]). There, we identify each cutpoint with the first location of the basic block that is included in the cutpoint set. To accommodate the SSA form, we identify each cutpoint with the first location *after* the ϕ-assignments (of the SSA form) of the basic block that is included in the cutpoint set. In Section 3 we describe the new computation of invariants, and in Section 4 we describe the new computation of the data abstraction mapping (which uses the computed invariants). In this section we demonstrate, by a simple example, how TVOC-SP computes the other ingredients of **Val**, and why it would fail to compute good invariants.

Consider the example in Fig. 2 which describes a sparse conditional constant propagation.

```
0:  B1 ← 1;
    W1 ← 1;
    Y1 ← 0;
1:  if (B1 = 1)
      N1 ← 500;
    else
      N2 ← 0;
2:  N3 ← φ(N1, N2);
3:  W3 ← φ(W1, W2);
    Y3 ← φ(Y1, Y2);
4:  if (W3 > N3) goto 7;
5:  W2 ← W3 + 2 * Y3 + 3;
    Y2 ← Y3 + 1;
6:  goto 3;
7:  return Y3;
8:
```

(a) Source

```
9:  w1 ← 1;
    t1 ← 0;
10: w3 ← φ(w1, w2);
    t3 ← φ(t1, t2)
11: w2 ← w3 + t3 + 3;
    t2 ← t3 + 2;
12: if (w2 ≤ 500) goto 10;
13: y1 ← t2/2;
    return y1;
14:
```

(b) Target

Fig. 2. An example of source and target

With the above modification to accommodate the SSA from, Tvoc-sp computes:

- CP_T, the target cutpoint set, consists of locations: 9, which is the first location in the initial block, 14, which is the terminal location, and 11, which is the first non-ϕ location in the basic block of the loop's body (that consists of locations 10, 11 and 12). Similarly, CP_S, the target cutpoint set, consists of locations $\{0, 5, 8\}$.
- The control mapping κ is $\{9 \mapsto 0, 11 \mapsto 5, 14 \mapsto 8\}$;
- The simple paths in the source are:

$$\{0 \to 5, 5 \to 5, 5 \to 8, 0 \to 8\}$$

  and the simple paths in the target are:

$$\{9 \to 11, 11 \to 11, 11 \to 14\}$$

- The transition relation for the source path $[5 \to 5]$ in P_S is:

$$\begin{aligned}(\mathtt{PC} = 5) \ \wedge \ (\mathtt{PC}' = 5) \ \wedge \ (W_2' = W_3 + Y_2 \cdot 2 + 3) \ \wedge \ (Y_2' = Y_3' + 1)\\ \wedge \ (W_3' = W_2') \ \wedge \ (Y_3' = Y_2') \ \wedge \ (W_3' \leq N_3)\\ \wedge \ \mathit{pres}(V_S - \{\mathtt{PC}, W_2, W_3, Y_2, Y_3\})\end{aligned}$$

where $\texttt{pres}(V)$ is an abbreviation of $\bigwedge_{v \in V} (v' = v)$

TVOC-SP computes invariants by fixed-point computations depending on reaching definitions. For this code, TVOC-SP cannot establish valid verification conditions since it cannot generate the invariant $N_3 = 500$ at location 5. The invariant computation presented in the next section can trivially detect this invariant.

3 Generating Invariants

Invariant generation is the most challenging task in the application of **Val**. In this section we describe a new invariant generation method, that operates on IR programs in SSA form. Since the data-flow based method currently implemented in TVOC is purely syntactic, and the new method is also semantic, we believe it to be both more efficient and to generate more information than the data-flow based method.

Intuitively, the role of invariants in **Val** is to carry information between basic blocks. For a program in SSA form, such a task can be performed by collecting the definitions, as well as the branching conditions, that reach certain program points, which can later be fed into a theorem prover with some arithmatics capabilities to reveal "hidden" information.

Consider the SSA program in Fig. 2 (a). Since the branch condition of the if-statement at `L1` always evaluates to *true*, N_3 evaluates to a constant value 500 at `L3`. Rather than attempting to directly detect $(N_3 = 500)$ as an invariant at `L3`, we backtrack from `L3` to collect data flow information that the definition of N_3 depends on. N_3 is either assigned to the value of N_1 or the value of N_2 depending on the branch condition $(B_1 = 1)$, thus we obtain

$$(N_3 = N_1 \ \wedge \ N_1 = 500 \ \wedge \ B_1 = 1) \ \vee \ (N_3 = N_2 \ \wedge \ N_2 = 0 \ \wedge \ \neg(B_1 = 1)) \quad (1)$$

as an invariant at `L3`. Besides, since `L0` dominates `L3` and the program is in SSA form, the definition at `L0` holds at `L3`. Thus, we obtain

$$(B_1 = 1) \ \wedge \ (W_1 = 1) \ \wedge \ (Y_1 = 0) \quad (2)$$

as invariant at `L3`. The conjunction of Eq. 1 and Eq. 2 implies the desired $(N_3 = 500)$.

3.1 Computing Invariants for programs in SSA Form

Assume a control flow graph (CFG) G of an SSA-formed program whose loops are all *natural* – strongly connected components with single entry locations. (If some of G's loops are not natural, *node splitting* can be used to transform the offensive loops.) We denote the entry node of a loop as a *loop header*. We assume that each loop header has a single incoming edge from outside of the loop. (If this is not the case, we introduce *preheader*s – new (initially empty) blocks placed just before the

header of a loop, such that all the edges that had previously led into the header from outside the loop lead into the preheader, and there is a single new edge from the preheader into the header.)

Thus, we assume that G is a CFG with each node being a basic block. Since all of G's loops are natural, G's edges can be partitioned into backward edges and forward edges. The nodes of G and the forward edges define a directed acyclic graph (DAG) that induces a partial order among the nodes.

Following the standard notation, given two nodes $x, y \in G$, we say that x *dominates* y if every path from the entry of G into y passes through x. We say that x is an *immediate dominator* of y if every dominator of y is either x or else is strictly dominated by x. The immediate dominator of a node y is denoted by $idom(y)$. For a node $y \in G$, we denote by $G_{idom(y)}$ the graph obtained from G by removing all edges that lead into $idom(y)$, and then removing all the nodes and edges that do no reach y.

For a node $x \in G$, let $\texttt{assign}(x)$ be the set of non ϕ-assignments in x. If x is not a loop header, we define:

$$\texttt{gen}(x) \;=\; \bigwedge_{(v:=exp)\in \texttt{assign}(x)} (v = exp).$$

The expression $\texttt{gen}(x)$ describes the invariants generated by x regardless of its environment.

For a node x with an immediate successor y, we denote by $\texttt{cond}(x, y)$ the condition under which the control transfers from x into y.

Let $x \in G$ be a node that is *not* a loop header. Assume that x has m_x predecessors $x'_1, \ldots, x'_{m_x}$. Let V_x denote the set of variables defined by ϕ-functions in x. Assume that for every $v \in V_x$, the definition of v in x by the ϕ-function is

$$v_{t^x_0(v)} \leftarrow \phi(v_{t^x_1(v)}, \ldots, v_{t^x_{m_x}}(v)).$$

Let $x \in G$ be now a node which is a loop header. Obviously, if x is reached through a back edge, one must consider, in addition to the definitions of the induction variables as expressed after the ϕ-functions, the information about their updates in the loop's body. If $v^x_i, \ldots, v^x_K$ are the basic induction variables of the loop whose header is x, where each induction variable v^x_i is initialized to b_i before entering the loop, and is incremented by c_i at each iteration, we define:

$$\texttt{induc}(x) \;=\; \exists \hat{v}_x \geq 0. \bigwedge_{i=1}^{K} (v^x_i = b_i + c_i \times \hat{v}_x) \tag{3}$$

where $\hat{v}_x$ is a new variable. I.e., $\hat{v}_x$ is a loop iteration count, and $\texttt{induc}(x)$ captures the values of the induction variables at *some* (possibly the 0^{th}) iteration of the loop. We shall return to the issue of dealing with (i.e., eliminating) the existential variables in Section 3.2.

In Fig. 3 we describe data flow equations to compute the assertions $\texttt{in}(x)$ and

$\texttt{out}(x)$ for every node $x \in G$. The former is an invariant at the beginning of x, after all the ϕ-functions, and the latter is an invariant at the end of x. The invariants $\texttt{in}(x)$ and $\texttt{out}(x)$ can be computed for every $x \in G$ simultaneously by forward traversal of the DAG induced by the forward edges of G.

$$
\texttt{in}(x,G) \;=\; \begin{cases}
\texttt{out}(idom(x),G) \;\wedge & \\
\qquad \texttt{cond}(idom(x),x) \;\wedge\; \texttt{induc}(x) & \text{if } x \text{ is a loop header,} \\
\texttt{out}(idom(x),G) \;\wedge & \\
\qquad \bigvee_{i=1}^{m_x} \left(\begin{array}{l} \texttt{out}(x'_i, G_{idom(x)} \;\wedge\; \texttt{cond}(x'_i, x) \\ \wedge\; \bigwedge_{v \in V_x} (v_{t_0^x(v)} = v_{t_i^x(v)}) \end{array} \right) & \\
\qquad\qquad \text{otherwise}
\end{cases}
$$

$$
\texttt{out}(x,G) \;=\; \begin{cases}
\texttt{in}(x,G) \;\wedge\; \texttt{gen}(x) & \text{if } x \in G, \\
\mathit{true} & \text{otherwise.}
\end{cases}
$$

Fig. 3. Data-flow equations for $\texttt{in}(x,G)$ and $\texttt{out}(x,G)$

Theorem 3.1 *The data-flow equations computed in Fig. 3 are sound, i.e., during an execution of a program, for every basic block* B *represented by node x in the* CFG *G, when the program reaches* B *(after the ϕ-functions),* $\texttt{in}(x,G)$ *holds, and whenever the program exits* B, $\texttt{out}(x,G)$ *holds.*

Proof Outline: The proof is by induction on the breadth first search of the G. The base case is for the entry node(s). Then, we have the data-flow equation for out, which is trivially true. For the inductive step, we distinguish between loop headers and non-loop headers. Suppose that x is a loop header. Since we assume that all loops are natural, the control reaches a loop header either from its immediate dominator or from a back edge. Since we assume SSA form, we have that all invariants at the end of the immediate dominator, as well as the condition leading to the loop, hold. If this is the first entry to the loop, then $\texttt{induct}(x)$ holds with the trivial $\hat{v_x} = 0$. If the loop header is reached by a back edge, then $\texttt{induct}(x)$ holds with the trivial $\hat{v_x} > 0$. By the induction hypothesis, the $\texttt{out}(idom(x),G)$ is sound. We can therefore conclude that $\texttt{in}(x,G)$ is sound.

If x is not a loop header, then x is reached through one of its predecessors, x'_i, with $\texttt{cond}(x'_i,x)$ holding. Thus, the soundness of $\texttt{in}(x,G)$ follows immediately from the induction hypothesis.

Finally, whether x is or is not a loop header, $\texttt{gen}(x)$ holds after block x is executed. Therefore, $\texttt{out}(x,G)$ is a conjunction of $\texttt{in}(x,G)$ with $\texttt{gen}(x)$. Since we assume that $\texttt{in}(x,G)$ is sound, the soundness of $\texttt{out}(x,G)$ follows. □

It thus follows from Theorem 3.1 that for every initial location i of the CFG corresponding to G, we can take $\varphi(i)$: $\texttt{in}(x,G)$ as the invariant at i.

Note that $\texttt{in}(x, G)$ and $\texttt{out}(x, G)$ are mutually recursive functions over the structure of the DAG induced by G. Hence, no fix-point computation is necessary to solve the equations. As a matter of fact, each node and edge of x's ancestor in G is visited only once in the computation of $\texttt{in}(x, G)$ and $\texttt{out}(x, G)$, thus the complexity of the computation is linear to the size of G.

3.2 Quantifier Elimination

Since the VCs generated by **Val** have invariants on both sides of implications, and since most theorem provers do not accept existential formulae as consequents, we need to eliminate such quantification in invariants. We accomplish this by instantiating the offensive existential quantifiers. In this discussion, we consider only the case of target invariants. The case of source invariants is similar.

Suppose a VC C_{ij} that has a $\exists \hat{v}_x$ on the right-hand-side. From the invariant generation algorithm, it follows that x is a loop header. Let "the loop" mean "the loop whose header is x" for this discussion. Assume that on the left-hand-side of C_{ij} has the invariant $\varphi_{_T}(i)$. We distinguish between the following cases:

j is the loop's header, and i is outside the loop. Thus, the simple path between i and j is one that corresponds to the first entry into the loop. In this case, we can instantiate $\hat{v}_x$ to 0, and replace the existential part of $\varphi'_{_T}(j)$ with

$$\bigwedge_{i=1}^{K}(v_i = b_i).$$

j is the loop header and i is in the loop. Thus, the simple path between i and j corresponds to a back edge of G. We can then "reuse" the value of $\hat{v}_x$ from the antecedent and replace the existential part of $\varphi'_{_T}(j)$ with

$$\bigwedge_{i=1}^{K}(v_i = b_i + c_i \times (\hat{v}_x + 1))$$

Neither of the previous cases. Thus, the simple path between i and j does not alter the values of the induction variables, and we can "reuse" the value of $\hat{v}_x$ from the antecedent, thus replacing the existential part of $\varphi'_{_T}(j)$ with

$$\bigwedge_{i=1}^{K}(v_i = b_i + c_i \times \hat{v}_x)$$

3.3 Example: Invariant Generation

To apply the method described above to the target program in our running example, we define basic blocks $\texttt{B1} = \{\texttt{L9}\}$, $\texttt{B2} = \{\texttt{L10}, \texttt{L11}, \texttt{L12}\}$, $\texttt{B3} = \{\texttt{L13}\}$, for which we

have:

$$\begin{aligned}
&\texttt{gen(B1)} &&: (t_1 = 0) \ \wedge \ (w_1 = 1)\\
&\texttt{gen(B2)} &&: (w_2 = w_3 + t_3 + 3) \ \wedge \ (t_2 = t_3 + 2)\\
&\texttt{cond(B1, B2)} &&: \mathit{true}\\
&\texttt{cond(B2, B3)} &&: \neg(w_2 \leq 500)
\end{aligned}$$

The program has a loop {B2} in which there is one induction variable, t_3, which is initialized to t_1 before the loop and is incremented by 2 at each iteration.

We therefore have:

$$\texttt{induct(B2)} \ = \ \exists \hat{v}_{_T}.(t_3 = t_1 + 2\hat{v}_{_T})$$

Solving the equations of Fig. 3, we obtain:

$$\begin{aligned}
&\texttt{in(B1)} \ &&: \mathit{true}\\
&\texttt{out(B1)} &&: (t_1 = 0) \ \wedge \ (w_1 = 1)\\
&\texttt{in(B2)} \ &&: (t_1 = 0) \ \wedge \ (w_1 = 1) \ \wedge \ \exists \hat{v}_{_T} : (t_3 = t_1 + 2\hat{v}_{_T})\\
&\texttt{out(B2)} &&: (t_1 = 0) \ \wedge \ (w_1 = 1) \ \wedge \ \exists \hat{v}_{_T} : (t_3 = t_1 + 2\hat{v}_{_T})\\
&&&\quad \wedge \ (w_2 = w_3 + t_3 + 3) \ \wedge \ (t_2 = t_3 + 2)\\
&\texttt{in(B3)} \ &&: (t_1 = 0) \ \wedge \ (w_1 = 1) \ \wedge \ \exists \hat{v}_{_T} : (t_3 = t_1 + 2\hat{v}_{_T})\\
&&&\quad \wedge \ (w_2 = w_3 + t_3 + 3) \ \wedge \ (t_2 = t_3 + 2) \ \wedge \ \neg(w_2 \leq 500)
\end{aligned}$$

We can therefore conclude that:

$$\begin{aligned}
&\varphi_{_T}(9) \ &&: \mathit{true}\\
&\varphi_{_T}(11) &&: (t_1 = 0) \ \wedge \ (w_1 = 1) \ \wedge \ \exists \hat{v}_{_T} : (t_3 = t_1 + 2\hat{v}_{_T})
\end{aligned}$$

Similarly, for the source program in our running example, we can compute that:

$$\begin{aligned}
&\varphi_{_S}(0) &&: \mathit{true}\\
&\varphi_{_S}(5) &&: (N_3 = 500) \ \wedge \ (Y_1 = 0) \ \wedge \ (W_1 = 1)\\
&&&\quad \wedge \ \exists \hat{v}_{_S} : (Y_3 = Y_1 + \hat{v}_{_S}) \ \wedge \ (W_3 \leq N_3)
\end{aligned}$$

Since $\hat{v}_{_S}$ is the iteration counter of the source loop, and $\hat{v}_{_T}$ is its counterpart in the target, we can safely include $(\hat{v}_{_S} = \hat{v}_{_T})$ in the antecedent of both verification conditions $C_{11,11}$ and $C_{11,14}$, which allows us to relate loop induction variables in source and target. Together with $\varphi_{_S}(5)$ and $\varphi_{_T}(11)$, we obtain that $t_3 = 2 \cdot Y_3$ holds at target cutpoint L11 (and its corresponding source cutpoint L5), implying the equality between observable variables Y_3 and y_1 upon termination.

4 Computing Data Abstraction

The data abstraction α maps, for each target cutpoint, a conjunction of equalities. Each equality is of the form $U = E_U$, where U is a source variables and E_U is an expression over target variables. The meaning of including the conjunct $U = E_U$ in $\alpha(i)$ is that when the control of the target is in location i, and, consequently, the control of the source is in $\kappa(i)$, the value of U in $k(i)$ is the value of E_U computed at the target. The more precise the invariants are, the more precise α can be. In this section we describe how to compute α given the computation of the φ's described above.

The algorithm described here assumes that some initial set of equalities of the type $U = E_U$ are given. In practice, such an initial set can be obtained from the symbol table of the compiler if one can access it (which one can, in the case of ORC), or by "reasonable" guesses, .e.g, by assuming that variables with the same name in source and target are equal. The correctness of the algorithm does not depend on the choice of the initial set, in particular, the initial set may include wrong equalities. Its ability to generate precise data abstraction, however, may be impacted by the initial set. We denote this initial set by Γ.

The algorithm is presented in Fig. 4. For each cutpoint i, the algorithm employs an auxiliary $\gamma(i)$ which is the set of equalities (of the from $U = E_U$). At each step, $\gamma(i)$ is the set of equalities that are assumed to hold whenever target is at location i. Initially, $\gamma(i)$ is Γ itself. At each iteration, until $\gamma(i)$ is stabilizes, equalities that violate the invariants and transition relations are removed. At each step, $\alpha(i)$ is the conjunction of the equalities in $\gamma(i)$. An equality $U = E_U$ is removed from $\gamma(i)$ if, for some simple path leading from j into i,

$$\varphi_T(j) \ \wedge \ \varphi_S(\kappa(j)) \ \wedge \ \alpha(j) \ \wedge \ \rho_\pi^T \ \wedge \ \rho_{\kappa(j)\kappa(i)}^S \quad \not\rightarrow \quad U' = (E_U)'$$

where $(E_U)'$ is the evaluation of E_U after the transition, i.e., E_U where every variable is replaced by its primed version.

Note that while we choose the target transition function that corresponds to the target path π, we take the matching source path to be any path in between the two matching π-endpoints of the source. Thus, if there are several matching source paths, we take the disjunction of their respective transition relations.

The procedure can be viewed as an iterative forward data-flow analysis, operating on the lattice $(2^\Gamma, \subseteq)$. The flow function of γ can be solved by starting with Γ and descending values in the lattice at each iteration, until a fixpoint is reached. The validity of the logical formula in the flow equation of γ is decided by using a theorem prover. Unlike iterative data flow analyses used in compiler optimizations, this procedure applies a joint analysis on source and target programs.

In our running example, we obtain in the data abstraction, e.g.,

$$\begin{aligned}
\alpha(9)&: \ (W_0 = w_0) \ \wedge \ (Y_0 = y_0)\\
\alpha(11)&: \ (W_3 = w_3)\\
\alpha(14)&: \ (W_3 = w_2) \ \wedge \ (Y_3 = y_1)
\end{aligned}$$

```
for each target cutpoint i
  γ(i) := Γ
  α(i) = ⋀_{E∈γ(i)} {E}

repeat
  for each target cutpoint i
    for every simple path π leading into i
        j := start point of π
        γ(i) :=
          {E ∈ γ(i) : φ_T(j) ∧ φ_S(κ(j)) ∧ α(j)  ∧  ρ^T_π ∧ ρ^S_{κ(j)κ(i)}  →  E'}
        α(i) = ⋀_{E∈γ(i)} E
until sets stabilize
```

Fig. 4. Computation of Data Abstraction

Note that had we started the algorithm with Γ containing only simple equalities of the form $(U = u)$, more complex data mappings would be captured with the aid of the source invariants. E.g., to express $\alpha(i)\colon (X = x) \wedge (Y = y) \wedge (Z = x + y)$, it suffices to obtain $\alpha(i)\colon (X = x) \ \wedge \ (Y = y)$ and $\varphi_S(\kappa(i))\colon (Z = X + Y)$.

5 Discussion and Conclusion

In this paper we propose a method based on SSA-from to compute invariants and data mappings. The method allows for translation validation of programs that are beyond the current power of TVOC. The translation to SSA-form and the new algorithms proposed here were implemented and tested.

```
B0:  a1 ← 3
     d1 ← 2
B1:  d3 ← φ(d1, d2)
     a3 ← φ(a1, a2)
     f1 ← a3 + d3
     g1 ← 5
     a2 ← g1 − d3
     d2 ← 2
     if (f1 ≤ g1) goto B1
B2:
```

(a)

```
B0:  a1 ← 1
     b1 ← 1
B1:  a3 ← φ(a1, a2)
     b3 ← φ(b1, b2)
     if (a3 mod 2 = 0) goto B3
B2:  a4 ← a3 + 1
     b4 ← b3 + 1
     goto B4
B3:  a5 ← a3 + 3
     b5 ← b3 + 3
B4:  a2 ← φ(a4, a5)
     b2 ← φ(b4, b5)
     if (b2 < n0) goto B1
```

(b)

Fig. 5. Examples where our invariant generation fails

Yet there are some optimizations for which we still fail to generate good invariants. Notable examples are optimizations that include sparse conditional constant folding and global value numbering: Consider the programs in Fig. 5. Clearly, in the code in (a), $a_i = 3$ and $d_i = 2$ are invariant in all cutpoints where a_i and d_i are defined. While the sparse conditional constant propagation algorithm of [13]

succeeds in determining this, our method, that does not propagate information iteratively, cannot. Similarly, in the code in (b), $a_i = b_i$ is invariant in all cutpoints where a_i and b_i are defined. While the value numbering algorithm of [1] succeeds in determining this, our method, being unable to proceed around loops, cannot.

Our technique may also be useful in proving properties of microcode. E.g., [2] describes a tool for proving backward compatibility of microcode. The microcode there is assumed not to contain loops. Our technique may allow for extending the work to microcode programs with loops. (In fact, our technique orignated with the work on the project reported on in [2].) In addition, we expect the techniques to be applicable to property verification of microcode programs.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[2] T. Arons, E. .Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck. Formal verification of backward compatibility of microcode. In *Proceedings of 17th International Conference on Computer Aided Verification (CAV 2005)*, number 3576 in Lncs, pages 185–200. Springer, 2005.

[3] C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. Tvoc: A translation validator for optimizing compilers. In *Proceedings of 17th International Conference on Computer Aided Verification (CAV 2005)*, Lncs, pages 291–295. Springer, 2005.

[4] S. Chan, R. Ju, and C. Wu. Tutorial: Open research compiler (orc) for the itanium processor family. In Micro 34, 2001.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM Press.

[6] R. Cytron, Ronald, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the program dependence graph. In *ACM TOPLAS*, volume 13, pages 451–490, New York, NY, USA, 1991. ACM Press.

[7] Y. Fang. *Translation Validation of Optimizing Compilers*. PhD thesis, NYU, http://cs.nyu.edu/web/Research/theses.html, 2005.

[8] 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.

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

[10] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. 4^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 151–166, 1998.

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

[12] Silicon Graphics. Whirl Intermediate Language Specification. www.cs.ualberta.ca/~amaral/courses/680/orc/whirl.pdf.

[13] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[14] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *Jounal of Universal Computer Science*, 9, 2003.

[15] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Journal on Formal Methods in System Design*, 27(3), 11 2005. Preliminary version appeared in *Proceedings of the Run-Time Result Verification Workshop*, Electronic Notes in Theoretical Computer Science (ENTCS), 2002, 70(4).