Validating More Loop Optimizations

Ying Hu Clark Barrett Benjamin Goldberg Amir Pnueli [1]

Department of Computer Science
New York University
New York, USA

Abstract

Translation validation is a technique for ensuring that a translator, such as a compiler, produces correct results. Because complete verification of the translator itself is often infeasible, translation validation advocates coupling the verification task with the translation task, so that each run of the translator produces verification conditions which, if valid, prove the correctness of the translation.

In previous work, the translation validation approach was used to give a framework for proving the correctness of a variety of compiler optimizations, with a recent focus on loop transformations. However, some of these ideas were preliminary and had not been implemented. Additionally, there were examples of common loop transformations which could not be handled by our previous approaches.

This paper addresses these issues. We introduce a new rule REDUCE for loop reduction transformations, and we generalize our previous rule VALIDATE so that it can handle more transformations involving loops. We then describe how all of this (including some previous theoretical work) is implemented in our compiler validation tool TVOC.

Key words: Translation validation, formal methods, loop optimizations

1 Introduction

Compiler correctness is essential to having correct programs. For critical applications, it is not enough to have a proof of correctness for the source code. There must also be an assurance that the compiler produces a correct translation of the source code into target machine code. Verifying the correctness of modern optimizing compilers is a challenging task because of their size, their complexity, and their evolution over time.

[1] Email: {yinghu,barrett,goldberg,amir}@cs.nyu.edu

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

Translation Validation (TV) [12] is a technique for ensuring that the target code emitted by a translator, such as a compiler, is a correct translation of the source code. Because of the difficulty of verifying an entire compiler, i.e. ensuring that it generates the correct target code for every acceptable source program, translation validation is used to validate each run of the compiler, comparing the actual source and target code.

In previous work [14,15], the proof rule Permute was introduced to validate loop reordering transformations and the proof rule Validate was introduced to validate so-called structure preserving transformations. However, we have since found these rules to be insufficient for certain kinds of transformations performed by optimizing compilers. For example, a loop which repeatedly increments a variable can be replaced by a single multiplication operation. For this kind of transformation, we introduce a new proof rule Reduce. In addition, we found that in some structure preserving cases involving nested loops, rule Validate is unsuccessful. However, by generalizing the rule slightly (obtaining a rule we call Gen-Validate), we can handle these cases as well.

TVOC is a tool being implemented at NYU which implements translation validation for the Intel Open Research Compiler (ORC) [7]. TVOC implements rules Permute, Gen-Validate, and Reduce and checks the generated verification conditions using the automatic theorem prover CVC Lite [1]. The latest version of TVOC also includes a concrete implementation of ideas suggested in [4] for validating reordering loop optimizations. In particular, TVOC now uses a uniform proof rule for all reordering loop optimizations, a heuristic for determining which loop optimizations occurred (rather than relying on the compiler for this information), and a methodology for combinations of optimizations.

This paper is organized as follows: Section 2 reviews transition systems which provide the necessary theoretical foundation for translation validation of optimizing compilers. Section 3 discusses related work on compiler verification in general and our previous work on translation validation of compilers in particular, including the rules Validate and Permute. Section 4 introduces the new proof rule Reduce. Section 5 proposes the proof rule Gen-Validate, a generalization of Validate. Finally, Section 6 describes how all of these proof rules are implemented in the latest version of TVOC.

2 Background

In order to discuss the formal semantics of programs, we briefly review *transition systems*, TS's, a variant of the *transition systems* of [12]. A *Transition System* $S = \langle V, \mathcal{O}, \Theta, \rho \rangle$ is a state machine consisting of: a set V of *state variables*; a set $\mathcal{O} \subseteq V$ of *observable variables*; an *initial condition* Θ, which is a formula over V characterizing the initial states of the system; and a *transition relation* ρ, a formula over 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 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. We assume that each transition system has a variable pc that describes the program location counter.

While it is possible to assign a transition relation to each statement separately, we prefer to use a *generalized* transition relation, describing the effect of executing several statements along a path of a program from one basic block to another. Consider the following piece of code:

```
L0:
    n := 500;
    y := 0;
    if (n < w) goto L2;
L1:
    ...
L2:
    ...
```

There are two disjuncts in the transition relation whose starting locations are $\mathtt{L}_0$. The first describes the $\mathtt{L}_0$ to $\mathtt{L}_1$ path, which is $\mathtt{pc} = \mathtt{L}_0 \ \wedge \ n' = 500 \ \wedge \ y' = 0 \ \wedge \ n' \geq w' \ \wedge \ \mathtt{pc}' = \mathtt{L}_1$, and the second describes the $\mathtt{L}_0$ to $\mathtt{L}_2$ path, which is $\mathtt{pc} = \mathtt{L}_0 \ \wedge \ n' = 500 \ \wedge \ y' = 0 \ \wedge \ n' < w' \ \wedge \ \mathtt{pc}' = \mathtt{L}_2$. The complete transition relation is formed by taking the disjunction of all such generalized transition relations.

The observable variables are the variables we care about, where we treat I/O devices as variables, and each I/O operation, including external procedure calls, removes elements from or appends elements to the corresponding variable. If desired, we can also include among the observable variables 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, i.e. are related by a one-to-one correspondence relation.

A computation of a TS is a maximal (possibly 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.

A transition system $\mathcal{T}$ is called *deterministic* if the observable part of the initial condition uniquely determines the rest of the computation. 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. Two such 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, then σ_T is terminating (finite) iff σ_S is and, in the case of termination, their final states are compatible. It is not hard to see that this notion of refinement is an equivalence relation. We will use $P_T \sim P_S$ to denote that P_T is a correct translation of P_S.

We distinguish *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, from *structure modifying* optimizations that admit no such clear mapping. Most high-level optimizations are structure preserving, while most loop optimizations are structure modifying.[2]

3 Previous and Related Work

3.1 Related Work

Traditional compiler verification tries to prove compiler correctness using standard program verification techniques. However, in practice this is often infeasible due to the complexity and evolution of compiler implementations. Recently, a number of creative techniques for verifying compilers have been introduced [5,6,8,9,10,11,12,13].

In [9,11], a certifying compiler provides the proof for type safety and memory safety properties of the target program, while our approach proves the semantic equivalence of the source and target program.

[10] verifies the preservation of semantics for each compilation and thus has the same goal as our work. Instead of using an automatic theorem prover, a set of algebraic rules are used to check the equivalence of logic formulas. The cases with branch splitting and loop optimizations are not handled there.

A *credible compiler* [13] produces an inductive proof along with each compilation, similar to our approach but with different algorithms and rules. However, the method proposed there assumes *full* instrumentation of the compiler, which is not assumed here or in [10].

In [5], the notion of correct translation and the method of program checking appear similar to ours. However, instead of transition systems, abstract state

[2] Some transformations such as skewing, unrolling, and peeling, can actually be handled by both our structure modifying and structure preserving proof approaches.

machines (ASMs) have been used there to model the operational semantics of programs, and their work does not deal with optimizations.

Comparison checking [6] is a technique that automatically checks the semantic equivalence of executions of source and target programs at *run-time*. Though it has the advantage of being precise, it cannot validate a program translation for all possible program inputs, and it increases the run-time of the program.

In [8], compiler optimizations are automatically proved correct using the automatic theorem prover Simplify [2]. Optimizations are proved once for all possible inputs so that the result is a verified compiler. However, the compiler writers have to use a domain-specific language called Cobalt and provide complicated rewrite rules with triggering guards. This approach also assumes that the compiler is written with verification in mind and that the transformations which have been verified are correctly implemented. We do not make these assumptions.

Our approach, translation validation [12], is similar to many of these approaches in that it focuses on verifying a single run of the compiler, rather than verifying the compiler itself. However, our work has the advantage that its abstract computational model and refinement concepts are very general. Also, it can be used to verify existing compilers due to its independence from the compiler. Finally, though it does require extra effort at compile time, it does not increase the run-time of programs.

3.2 Previous Work

3.2.1 Rule Validate *for structure preserving optimizations*

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 comparable TS's, where P_S is the *source* and P_T is the *target*. In order to establish that P_T is a correct translation of P_S for the cases that the structure of P_T does not radically differ from the structure of P_S, we use a proof rule, Validate, which is inspired by the computational induction approach ([3]), originally introduced for proving properties of a single program. Rule Validate (see [14], and a variant in [15] which produces simpler verification conditions) provides a proof methodology by which one can prove that one program *refines* another. This is achieved by establishing a *control mapping* from target to source locations, a *data abstraction* mapping from source variables to (possibly guarded) expressions over the target variables, and proving that these abstractions are maintained along basic execution paths of the target program.

In Validate, each TS is assumed to have a *cut-point* set, a subset of the program locations (i.e. possible values of $\mathtt{pc}$) that includes all initial and terminal locations, as well as at least one location from each of the cycles in the program's control flow graph. A *simple path* is a path connecting two cut-points, and containing no other cut-point as an intermediate node. For each simple path, we can (automatically) construct the transition relation of the path. Such a transition relation contains the condition (if any) which enables

this path to be traversed and the data transformation effected by the path.

Rule Validate constructs a set of verification conditions, one for each simple target path, whose aggregate consists of an inductive proof of the correctness of the translation between source and target. Roughly speaking, each verification condition states that, if the target program can execute a simple path, starting with some conditions correlating the source and target programs, then at the end of the execution of the simple path, the conditions correlating the source and target programs still hold. The conditions consist of the control mapping, the data mapping, and, possibly, some invariant assertion holding at the target code. Rule Validate is discussed in more detail in Section 5.

3.2.2 Translation Validation of Reordering Loop Transformation

We described in [14] and [4] our approach to performing translation validation of loop reordering transformations, such as loop interchange, fusion, distribution, and tiling. This previous work, in which the proof rule Permute was described, relies on recognizing which loop optimizations have been performed. In this paper, in section Section 6.1, we describe the simple heuristic algorithm that we have implemented in TVOC to recognize which optimizations have been performed.

4 Rule Reduce

$$\textbf{for } i = 1 \textbf{ to } N \textbf{ do } x := x + 1; \quad \Longrightarrow \quad x := x + N;$$

Fig. 1. An example for loop reduction.

Rule Permute can handle any loop reordering transformation, but there are other kinds of loop transformations that cannot be handled by either Validate or Permute. Fig. 1 shows an example (an actual transformation performed by ORC) in which a loop is removed and replaced with a single statement. We call this **loop reduction** and propose a new proof rule, Reduce, to deal with such cases. Rule Reduce is shown in Fig. 2, where the symbol $\sim$ means that two pieces of code are equivalent.

$$\begin{array}{ll} \text{R1.} & B(1) \sim B'(1) \\ \text{R2.} & \forall i > 0 : B'(i); B(i+1) \sim B'(i+1) \end{array}$$

$$\textbf{for } i = 1 \textbf{ to } N \textbf{ do } B(i) \quad \sim \quad B'(N)$$

Fig. 2. Rule Reduce for loop reduction.

Loop reduction is based on finding a closed-form expression for the result of executing the loop. Such transformations can often be verified using induction. Rule Reduce is based on an inductive argument that executing $B(i)$ from 1 to N is equivalent to executing some closed-form block $B'(N)$. The first premise is the base case. It requires that $B(1)$ be equivalent to $B'(1)$. The second premise is the inductive case, which requires that $B'(i)$ be able to "absorb" $B(i+1)$ to become $B'(i+1)$. For the code in Fig. 1, $B(i)$ is $x := x + 1$ and $B'(i)$ is $x := x + i$. The two premises can easily be established for this simple case.

Rule Reduce can also be used to show that a loop which does nothing can be removed. Fig. 3 shows a transformation which removes a loop with no loop body. In this case, $B(i) = B'(i) =$ **Skip**.

for $i = 1$ **to** N **do**
  Skip;
$\Longrightarrow$
Skip;

Fig. 3. Reduction for an empty loop.

5 A Generalization of Rule Validate

Section 3 briefly described the proof rule Validate. Rule Validate can validate many transformations in which the source and the target have the same loop structure. However, there are still some cases in which, even though the loop structure is the same, rule Validate is unsuccessful. Fig. 4 gives an example of such a transformation performed by ORC.

$\mathtt{CP}_1$:
  for $i = 1$ **to** N **do**
$\mathtt{CP}_2$:
    for $j = 1$ **to** M **do**
$\mathtt{CP}_3$:
      $B(i, j)$;
$\mathtt{CP}_4$:

$\Longrightarrow$

$\mathtt{cp}_1$:
  if $(1 \leq N)$ **then** {
$\mathtt{l}_1$:
    if $(1 \leq M)$ **then** {
      for $i = 1$ **to** N **do**
$\mathtt{cp}_2$:
        for $j = 1$ **to** M **do**
$\mathtt{cp}_3$:
          $B(i, j)$;
    }
  }
$\mathtt{cp}_4$:

Fig. 4. An example for which rule Validate fails.

The transformation adds two "short-cut" branch conditions before the main loops. In this example, $\mathtt{CP}_1, \mathtt{CP}_2, \mathtt{CP}_3$ and $\mathtt{CP}_4$ are the source cut-points, and $\mathtt{cp}_1, \mathtt{cp}_2, \mathtt{cp}_3$ and $\mathtt{cp}_4$ are the target cut-points. The control mapping maps each of the target cut-points in order to the corresponding source cut-point.

The label $\mathtt{l}_1$ labels a target location that is not in the cut-point set. Now, consider a simple target path from $\mathtt{cp}_1$ to $\mathtt{cp}_4$. This path goes from $\mathtt{cp}_1$ through $\mathtt{l}_1$ and then to $\mathtt{cp}_4$ directly without ever entering the loops. This target path is enabled under the condition $N \geq 1 \wedge M < 1$. Its corresponding source path goes from $\mathtt{CP}_1$ inside the loop to $\mathtt{CP}_2$, stays at $\mathtt{CP}_2$ for N cycles, and then exits to $\mathtt{CP}_4$ without entering the inner loop. Since this source path crosses $\mathtt{CP}_2$ N times on its way from $\mathtt{CP}_1$ to $\mathtt{CP}_4$, it is not a simple path. This is a problem for rule Validate: the simple path from $\mathtt{cp}_1$ to $\mathtt{cp}_4$ has no corresponding simple path in the source! As a result, the verification condition corresponding to the simple path from $\mathtt{cp}_1$ to $\mathtt{cp}_4$ fails.

The reason that rule Validate fails for the transformation of Fig. 4 is that it assumes each simple path in the target corresponds to one or more simple paths in the source. However, this transformation transforms a non-simple path in the source into a simple path in the target. We can solve this problem by relaxing the requirement on the set of cut-points used by rule Validate.

The modified proof rule, Gen-Validate, is presented in Fig. 5, and a proof of its correctness is given in the appendix. It is essentially the same proof rule as that given in [15] except that a new item 0 has been added which explicitly allows the set of cut-points to be chosen more freely. The cut-point sets must include the initial and terminal points of programs as before, but they do not necessarily contain a point for each loop. Instead, we require that the transition relation for every simple path be "computable". Here, "computable" means that the path is finite and its transition relation can be calculated by data flow analysis or derived by proof rules. It is easy to see that loop-free paths are guaranteed to be computable. But it is also the case that whenever the number of iterations of a loop are known, the transition relation for the loop can be computed by unrolling the loop.

To solve the example of Fig. 4, we can eliminate cut-points $\mathtt{CP}_2$ and $\mathtt{cp}_2$ as shown in Fig. 6. There are now several new simple paths that did not exist before. Most of these are loop-free and are thus easily computable. However, there is now a new source path from $\mathtt{CP}_1$ to $\mathtt{CP}_4$. This path is only possible if the inner loop is never executed (otherwise $\mathtt{CP}_3$ would be reached). But this means that the loop body is effectively empty, and as discussed earlier (see Fig. 3), a loop with an empty body is equivalent to doing nothing. Note that such a path in the target is not feasible since it would require both $1 \leq M$ and $1 > M$ to be true. Thus, all of these paths are computable and the requirements for rule Gen-Validate are met. With this new set of cut-points, the validation succeeds because there is a corresponding simple source path for the target path from $\mathtt{cp}_1$ to $\mathtt{cp}_4$.

6 New Implementation Features

TVOC is a translation validation tool being developed at NYU. It accepts as

0. Establish source and target cut-point sets CP_S and CP_T, which include all initial and terminal program locations. For any simple path between two cut-points i and j, its transition relation ρ_{ij} must be computable.

1. Establish a *control abstraction* $\kappa\colon \mathsf{CP}_T \to \mathsf{CP}_S$ such that i is an initial (terminal) location of T iff $\kappa(i)$ is an initial (terminal) location S.

2. For each cut-point i in CP_T, form an *invariant* φ_i that may refer only to target variables.

3. Establish a *data abstraction*

$$\alpha : (\mathtt{PC} = \kappa(\mathtt{pc}) \ \wedge \ (p_1 \to V_1 = e_1) \ \wedge \ \cdots \ \wedge \ (p_n \to V_n = e_n)$$

which asserts that the source and target are at corresponding cut-points and which assigns to *some* non-control source variables $V_i \in V_S$ an expression e_i over the target variables, conditional on the (target) boolean expression p_i. It is required that for every initial target cut-point i, $\Theta_S \ \wedge \ \Theta_T \to \alpha \ \wedge \ \varphi_i$. It is also required that every *observable* source variable $V \in \mathcal{O}_S$ has a unique corresponding *observable* target variable $v \in \mathcal{O}_T$, and that for every terminal target cut-point t, $\mathtt{pc} = t \wedge \alpha$ implies that $V = v$ for all $V \in \mathcal{O}_S$.

4. For each pair of cut-points $i, j \in \mathsf{CP}_T$ such that there is a simple path from i to j, we form the verification condition

$$C_{ij}: \qquad \varphi_i \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho}_{ij} \ \wedge \ \Big(\bigvee_{\pi \in Paths(\kappa(i))} \overset{S}{\rho}_{\pi} \Big) \quad \to \quad \alpha' \ \wedge \ \varphi'_j,$$

where $Paths(\kappa(i))$ is the set of all simple source paths starting at $\kappa(i)$ and $\overset{S}{\rho}_{\pi}$ is the transition relation for the simple source path π.

5. Establish the validity of all the generated verification conditions.

Fig. 5. The generalized rule Gen-Validate

input a source program S and target program T. These are provided in the WHIRL intermediate representation, a format used by Intel's Open Research Compiler (ORC) [7] among others. The output of TVOC is either "VALID" or "INVALID", depending on whether the target is a correct translation of the source. In [4], a number of ideas were introduced which had not yet been fully developed or implemented in TVOC. We have since modified TVOC to implement these features as well as those described in this paper.

Fig. 7 shows the old architecture of TVOC. Three limitations of this architecture were addressed in [4]: first, we depended on an auxiliary file (the ".l" file) generated by the compiler to tell us which loop transformations had

```
CP_1 :
    for i = 1 to N do
       for j = 1 to M do
CP_3 :                                  ⟹
            B(i, j);
CP_4 :
```

```
cp_1 :
    if (1 ≤ N) then {
       if (1 ≤ M) then {
          for i = 1 to N do
             for j = 1 to M do
cp_3 :
                B(i, j);
       }
    }
cp_4 :
```

Fig. 6. Example with modified cut-points.

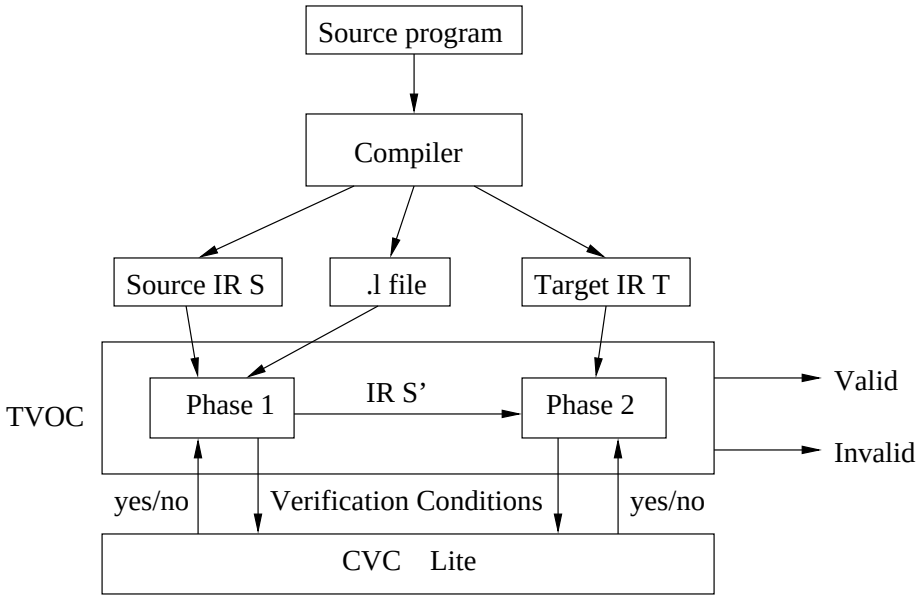

Fig. 7. The old architecture of TVOC.

been performed; second, loop transformations had to be verified in one step, even if the transformations were more naturally modeled as the composition of several simple transformations; and third, there were several different proof rules (and corresponding code) for verifying the loop transformations. In this section, we discuss the new architecture of TVOC (shown in Fig. 8) and show how we implemented the solutions to these problems. We also discuss the implementation of the Gen-Validate rule described in this paper.

6.1 An Algorithm for Inferring Loop Optimizations

Because it is nontrivial to figure out what kind of optimizations the compiler performs, the old version of TVOC used information produced by the compiler to figure out which loop optimizations had occurred. However, not all compilers provide such information, and the information provided by ORC was

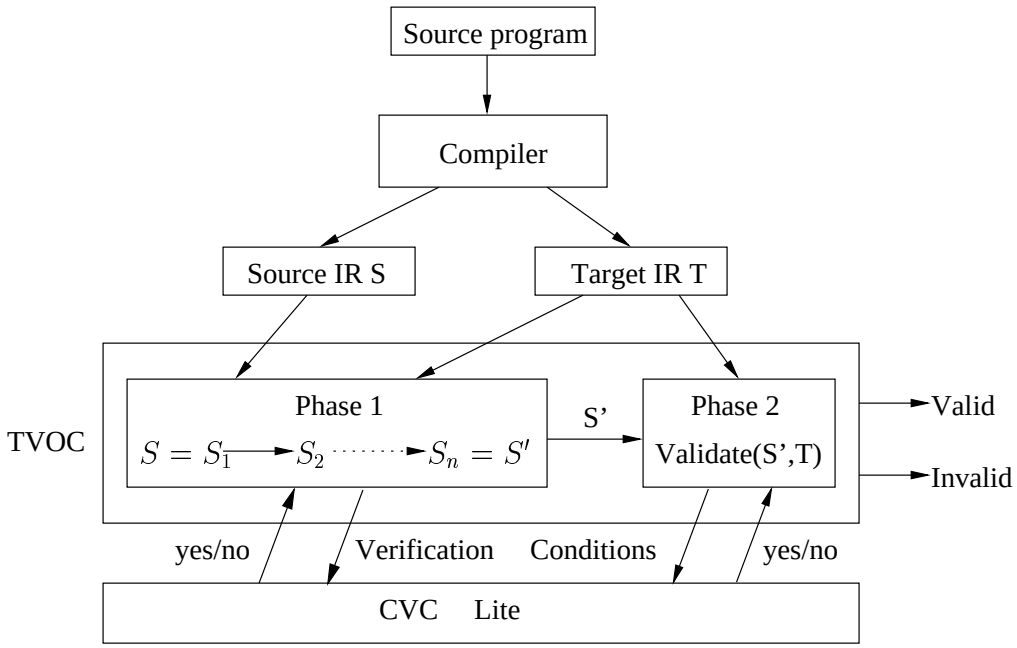

Fig. 8. The new architecture of TVOC.

1. For each nested loop of depth m in the source, check the corresponding target code for a nested loop of depth n. Note that we can match up loops in the source and target because WHIRL includes annotations indicating which line number in the source corresponds to a given target line.
2. If $m > n$, check whether the target contains code which came from the body of the source loop. If not, try loop reduction. Otherwise, try loop fusion.
3. If $m = n$, check to see if any indices are out of order. If so, loop interchange has occurred.
4. If $m < n$, assume loop tiling has occurred.

Fig. 9. The algorithm for analyzing loop transformations.

sometimes incomplete. In order to make TVOC more generally applicable, we developed an algorithm to infer which loop transformations were performed by looking only at the source and target code. The algorithm is capable of inferring loop reduction, loop fusion, loop interchange, and loop tiling and is shown in Fig. 9.

6.2 A Unified Validation Module for Reordering Optimizations

In previous versions, TVOC used different proof rules for interchange and tiling than it does for fusion, and it had different modules for different loop transformations. This was not ideal from a software engineering perspective. In the current version, TVOC uses the generalized approach described in [4] for all reordering loop transformations. Thus, there is only one general module for checking reordering transformations which accepts the loop index domain and permutation function as parameters and generates the appropriate verification conditions.

6.3 A Methodology for Combinations of Optimizations

The old version of TVOC had difficulty handling combinations of loop transformations. This was a serious drawback since often multiple transformations are performed by the compiler. In the new version, after a loop transformation is inferred and validated, TVOC synthesizes a new intermediate version of the code obtained by applying that transformation. It repeats this process for each detected transformation. In this way, a sequence $S_1, S_2, ...S_n$ of intermediate versions of the code is generated by TVOC, and the final version S_n is output by phase one and provided as input to phase two which uses the validate rule to check it against the target code.

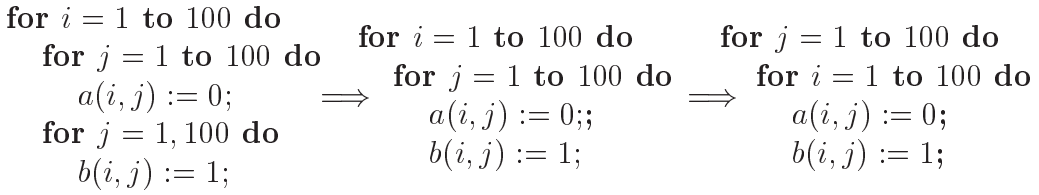

Fig. 10. A combination of loop transformations.

As an example, consider the code in Fig. 10. ORC first fuses the two inner loops and then performs loop interchange in order to improve cache performance (when the input code is written in Fortran in which arrays are stored in column major order). In phase 1, after comparing the source and target loops, TVOC detects that loop fusion and interchange happened. It first checks if fusion is valid. When the result is positive, it performs fusion and generates a new intermediate version of the code. Next it checks whether interchange is valid (which it is), generates a new intermediate version, and sends the result to phase two.

6.4 Implementation of Rule Gen-Validate

Recall that rule Gen-Validate requires checking a verification condition for each simple path between cut-points in the target. Furthermore, the verification condition for the target path from i to j includes a disjunction of all

possible simple source paths starting from $\kappa(i)$. In the actual implementation of rule Gen-Validate, it is not practical to test all the paths starting from cut-point $\kappa(i)$ in the source. It is much easier to restrict the source paths to those from $\kappa(i)$ to $\kappa(j)$. With some additional work, we can restrict our attention to only these paths.

The following theorem shows how to recast the verification condition in terms of only those source paths from $\kappa(i)$ to $\kappa(j)$. Let $Cond^S_\pi$ be the conditions under which a simple source path π is enabled (this corresponds to a conjunction of the branch conditions along the path).

Theorem 6.1 *Consider the following verification conditions:*

$$\varphi_i \ \wedge \ \alpha \ \wedge \ \rho^T_{ij} \ \wedge \ (\bigvee_{\pi \in Paths(\kappa(i))} \rho^S_\pi) \quad \rightarrow \quad \alpha' \ \wedge \ \varphi'_j, \tag{1}$$

$$\varphi_i \ \wedge \ \alpha \ \wedge \ \rho^T_{ij} \quad \rightarrow \quad (\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} Cond^S_\pi) \tag{2}$$

$$\varphi_i \ \wedge \ \alpha \ \wedge \ \rho^T_{ij} \ \wedge \ (\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} \rho^S_\pi) \quad \rightarrow \quad \alpha' \ \wedge \ \varphi'_j, \tag{3}$$

We claim that equation (1) holds iff (2) and (3) hold.

Proof. In one direction, suppose (1) holds, (3) also holds because the left-hand side of (3) is stronger than the left-hand side of (1) while the right-hand sides of the two implications are the same. Now, to show that (2) also holds, suppose we have $\varphi_i \ \wedge \ \alpha \ \wedge \ \rho^T_{ij}$. By definition of ρ and α, it follows that $\mathtt{PC} = \kappa(i)$. Since i is not a target terminal cut-point, $\kappa(i)$ is not a source terminal cut-point. Now, at every non-terminal source cut-point, some transition must be taken, so it follows that $(\bigvee_{\pi \in Paths(\kappa(i))} \rho^S_\pi)$ holds. By (1), we then have $\alpha' \ \wedge \ \varphi'_j$. But from $\rho^T_{ij} \wedge \alpha'$, $\mathtt{PC}' = \kappa(j)$ follows. We thus have $(\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} \rho^S_\pi)$, which means (2) holds since $Cond^S_\pi$ is implied by ρ^S_π.

In the other direction, suppose that (2) and (3) hold and that we have $\varphi_i \ \wedge \ \alpha \ \wedge \ \rho^T_{ij} \ \wedge \ (\bigvee_{\pi \in Paths(\kappa(i))} \rho^S_\pi)$. By (2), we have $(\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} Cond^S_\pi)$, so some path π from $\kappa(i)$ to $\kappa(j)$ is enabled. But because the transition system is deterministic, only one path can be enabled at a given point, which means that at $\kappa(i)$, the only simple path enabled is from $\kappa(i)$ to $\kappa(j)$. Therefore $(\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} \rho^S_\pi)$ holds. By (3), we have $\alpha' \ \wedge \ \varphi'_j$, and thus (1) holds. □

Using this theorem, we were able to implement part 4 of rule Gen-Validate by checking the conditions of the source simple paths between $\kappa(i)$ and $\kappa(j)$ without looking for all the source simple paths starting from $\kappa(i)$.

7 Appendix: Soundness of Gen-Validate

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, where P_S is the *source* and P_T is the *target*. Assume all the parts in rule Gen-Validate are established. We need to prove that P_T is a correct translation of P_S, which means 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.

From part 3 of rule Gen-Validate, we know that the two systems are comparable. We will prove the rest in two directions.

Suppose we have a terminating target computation σ_T. We know that the initial state t_0 and terminal state t_n of the computation must be at some target cut-points cp_0 and cp_n, according to part 0 of rule Gen-Validate. According to part 1 of Gen-Validate, the corresponding source cut-points CP_0 and CP_n are initial and terminal source cut-points respectively, and for any other cut-point cp_i in the target computation path, the corresponding source cut-point CP_i is $\kappa(cp_i)$. Now, by part 3, $\alpha \wedge \phi$ holds at the initial states t_0 and s_0. From part 4, for any cut-point i and its next cut-point j in the target path,

$$C_{ij}: \qquad \varphi_i \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho}_{ij} \ \wedge \ \Big(\bigvee_{\pi \in Paths(\kappa(i))} \overset{S}{\rho}_{\pi} \Big) \quad \rightarrow \quad \alpha' \ \wedge \ \varphi'_j.$$

Here, since the source cut-point $\kappa(i)$ is not the terminal cut-point, there is always a source path enabled at $\kappa(i)$, which means $\bigvee_{\pi \in Paths(\kappa(i))} \overset{S}{\rho}_{\pi}$ is always true. This condition guarantees that for the target simple path between i and j (it has computable transition relation $\overset{T}{\rho}_{ij}$, and its corresponding source simple path also has a computable transition relation $\overset{S}{\rho}_{\pi}$), if $\alpha \wedge \phi$ holds at cut-points cp_i and $\kappa(cp_i)$, then it also holds at cp_j and $\kappa(cp_j)$. By induction, it follows that $\alpha \wedge \phi$ holds at the terminal cut-points, which have the states s_n and t_n. But by 3, this implies that s_n and t_n are compatible.

For the other direction, suppose we have a terminating source computation σ_S. Now, suppose the corresponding target computation σ_T is non-terminating. This infinite target path will include an infinite number of target cut-points, since it is required that the transition relation for the path between two directly connected cut-points be computable and only a finite path can have a computable transition relation. By the argument above, a target computation with an infinite number of cut-points will have a corresponding source computation σ'_S with an infinite number of source cut-points. This would require there to be two different source computations σ_S and σ'_S starting from the same initial source state s_0, which violates the assumption that the source program is deterministic. Therefore, the corresponding target computation σ_T must be terminating. And according to the previous argument, their final states must be compatible.

References

[1] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, July 2004. To appear.

[2] D. Detlefs, G. Nelson, and J. Saxe. Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, Systems Research Center, HP Laboratories, Palo Alto, CA, July 2003.

[3] R. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19:19–32, 1967.

[4] B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. In *Third International Workshop on Compiler Optimization meets Compiler Verificaiton (COCV)*, Apr. 2004.

[5] G. Goos and W. Zimmermann. Verification of compilers. *Lect. Notes in Comp. Sci.*, 1710:201–230, 1999.

[6] C. Jaramillo, R. Gupta, and M. Soffa. Debugging and testing optimizers through comparison checking. *Lect. Notes in Comp. Sci.*, 65(2), 2002.

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

[8] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[9] G. Necula. Proof-carrying code. In *POPL'97*, pages 106–119, 1997.

[10] 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.

[11] G. 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.

[12] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*, pages 151–166, 1998.

[13] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, 1999.

[14] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. A translation validator for optimizing compilers. *Journal of Universal Computer Science*, 2003. Preliminary version in *ENTCS*, 65(2), 2002.

[15] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Journal of Formal Methods in System Design*, 2004. To appear, preliminary version in *ENTCS*, 70(4), 2002.