# The Theory for Validating Loop Optimizations

Ying Hu

Department of Computer Science
New York University
yinghu@cs.nyu.edu

[1]

**Abstract.** Translation Validation is a technique for ensuring that a translator 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 applied to give a framework for proving the correctness of a variety of compiler optimizations, with a recent focus on loop transformations. In the framework, a set of proof rules were proposed to establish the equivalence between the original and translated codes. However, these rules are too conservative, and they sometimes lead to false alarms in valid situations. Additionally, there were examples of common loop transformations which could not be handled by the previous theory.
This paper addresses these issues. We introduce an improved permutation proof rule INV-PERMUTE which considers the initial conditions and invariant conditions of the loop, a new rule REDUCE for loop reduction transformations, and we generalize our previous rule VALIDATE so that it can handle transformations which eliminate simple loops. We also give examples to illustrate these rules.

## 1 Introduction

Translation Validation (TV) 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 codes.

There has been considerable work [18,16,22,23,14] in this area to develop TV techniques for optimizing compilers that utilize *structure preserving* transformations, i.e. transformations which do not greatly change the structure of the program (e.g. dead code elimination, loop-invariant code motion, copy propagation) [1,20] as well as *structure modifying* transformations, such as reordering loop transformations (e.g. interchange, tiling), that do significantly change the structure of the program [2,21].

For the translation validation of reordering loop transformations, a proof rule PERMUTE was proposed in [23,4,7], which treats loop transformations as permutations. Although the PERMUTE rule has been used to check the validity of a number of reordering loop transformations, it has the limitation of requiring the loop transformations to be valid in all contexts without considering any conditions outside of the loop. In this paper, we introduce an improved permutation proof rule INV-PERMUTE which considers the context of the loop and thus is more powerful than the PERMUTE rule.

For the translation validation of structure preserving transformations, a proof rule VALIDATE was introduced in [22,23]. However, we have 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.

[1] Proceedings of MASPLAS'05
Mid-Atlantic Student Workshop on Programming Languages and Systems
University of Delaware, April 30, 2005

This paper is organized as follows. Section 2 discusses related work on compiler verification in general and our previous work on translation validation of optimizing compilers in particular, including the rules VALIDATE and PERMUTE. Section 3, Section 4 and Section 5 describes the new proof rules INV-PERMUTE for reordering loop transformations, REDUCE for loop reduction, and GEN-VALIDATE, the generalized rule for VALIDATE. Finally, Section 6 concludes.

## 2 Previous and Related Work

### 2.1 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 [8,11,13,15,16,17,18,19].

In [15,17], 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.

[16] 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* [19] 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 [16].

In [8], the notion of correct translation and the method of program checking appear similar to ours. However, instead of transition systems, abstract state machines (ASMs) have been used there to model the operational semantics of programs, and their work does not deal with optimizations.

Comparison checking [11] 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 [13], compiler optimizations are automatically proved correct using the automatic theorem prover Simplify [5]. 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 [18], 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.

### 2.2 Previous Work

#### 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 $\mathsf{TS}$'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 ([6]), originally introduced for proving properties of a single program. Rule VALIDATE (see [22], and a variant in [23] 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 $\mathsf{TS}$ 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.

### Rule PERMUTE for reordering loop transformations

Loop optimizations are often used by a modern compiler to improve parallelism and make efficient use of the memory hierarchy. Many loop transformations, including reversal, fusion, distribution, interchange, and tiling are *reordering* loop transformations, changing the iteration order of individual statements, but not changing which statements are executed. Since these transformations change the structure of the code significantly, they typically cannot be validated by rule VALIDATE. Consider a general loop transformation:

$$\textbf{for } \vec{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\vec{i}) \quad \Longrightarrow \quad \textbf{for } \vec{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\vec{j})) \tag{1}$$

in which the loop index vector is changed from $\vec{i}$ to $\vec{j}$, the loop index domain is changed from $\mathcal{I}$ to $\mathcal{J}$, the iteration order is changed from $\prec_{\mathcal{I}}$ to $\prec_{\mathcal{J}}$, and $F$ is a bijection from $\mathcal{J}$ to $\mathcal{I}$. Here, $B$ is the loop body and is parameterized by the loop index vector.

In [22], rule PERMUTE was proposed for validating such loop reordering transformations. As shown in Fig. 1, there are two requirements that must be satisfied to verify a reordering transformation: the mapping $F$ must be a bijection from $\mathcal{J}$ onto $\mathcal{I}$, and for every pair of loop index vectors $\vec{i}_1, \vec{i}_2$, such that the order of execution of $\mathtt{B}(\vec{i}_1)$ and $\mathtt{B}(\vec{i}_2)$ is reversed after the transformation, the result of executing the pair of iterations in either order must be the same.

$$\begin{array}{lll}
\text{R1. } \forall \vec{i} \in \mathcal{I} : \exists \vec{j} \in \mathcal{J} : & \vec{i} = F(\vec{j}) & \\
\text{R2. } \forall \vec{j}_1 \neq \vec{j}_2 \in \mathcal{J} : & F(\vec{j}_1) \neq F(\vec{j}_2) & \\
\text{R3. } \forall \vec{i}_1, \vec{i}_2 \in \mathcal{I} : & \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \wedge F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1) \implies & \mathtt{B}(\vec{i}_1);\mathtt{B}(\vec{i}_2) \sim \mathtt{B}(\vec{i}_2);\mathtt{B}(\vec{i}_1) \\
\hline
\multicolumn{3}{c}{\textbf{for } \vec{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\vec{i}) \quad \sim \quad \textbf{for } \vec{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\vec{j}))}
\end{array}$$

**Fig. 1.** Rule PERMUTE for Reordering Transformations

Rule PERMUTE can not only deal with transformations that change the execution order of a single loop body, but also handle a wide variety of loop structures including multiple and nested loops by considering more sophisticated domains for $\mathcal{I}$ [7]. However, rule PERMUTE is limited to reordering transformations that are valid in all contexts. In Section 3, a more general proof rule INV-PERMUTE will be presented which is able to consider the initial and invariant conditions of the loop.

## 3 Rule INV-PERMUTE

We reviewed rule PERMUTE in Section 2. The rule is easy to implement, because it only needs the information inside the loop to generate the logical formula for code equivalence, without explicitly having to perform dependence analysis, and it can leave the task of proving the legality of transformations to an automatic theorem

```
for i = 1 to N                                        for j = 1 to M
   for j = 1 to M                        ⟹              for i = 1 to N
      A[i+k, j+1] = A[i, j] + 1                            A[i+k, j+1] = A[i, j] + 1
```

**Fig. 2.** A loop interchange example

prover. However, rule PERMUTE does not take the context of a loop into account. The rule assumes that the program is in an arbitrary state, which requires premise 3 in Fig. 1 to be valid for all values of non-index variables. Consider the loop in Fig. 2, where loop interchange is invalid according to the PERMUTE rule. Notice that if $k$ happens to have a non-negative value upon entering the loop, then loop interchange *is* valid. From this example, we see that PERMUTE can be improved by incorporating a loop invariant $\phi$ (such as $k \geq 0$), so that premise 3 becomes:

$$\forall \vec{i}_1, \vec{i}_2 \in \mathcal{I}: \quad \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \ \wedge \ F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1) \quad \Longrightarrow \quad \{\phi\}\ \mathtt{B}(\vec{i}_1);\mathtt{B}(\vec{i}_2) \ \sim \ \{\phi\}\ \mathtt{B}(\vec{i}_2);\mathtt{B}(\vec{i}_1)$$

where the representation $\{\phi\}$ uses Hoare's precondition notation [9], meaning that we assume $\phi$ holds before each of the two pieces of code.

$$\begin{array}{lll} \text{R1. } \forall \vec{i} \in \mathcal{I}: \exists \vec{j} \in \mathcal{J}: & \vec{i} = F(\vec{j}) & \\ \text{R2. } \forall \vec{j}_1 \neq \vec{j}_2 \in \mathcal{J}: & F(\vec{j}_1) \neq F(\vec{j}_2) & \\ \text{R3. } \forall \vec{i} \in \mathcal{I}: & \{\phi\} \quad \mathtt{B}(\vec{i}) \quad \{\phi\} & \\ \text{R4. } \forall \vec{i}_1, \vec{i}_2 \in \mathcal{I}: & \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \ \wedge \ F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1) & \Longrightarrow \quad \{\phi\}\ \mathtt{B}(\vec{i}_1);\mathtt{B}(\vec{i}_2) \ \sim \ \{\phi\}\ \mathtt{B}(\vec{i}_2);\mathtt{B}(\vec{i}_1) \\ \hline \multicolumn{3}{c}{\{\phi\}\ \textbf{for}\ \vec{i} \in \mathcal{I}\ \textbf{by}\ \prec_{\mathcal{I}}\ \textbf{do}\ \mathtt{B}(\vec{i}) \quad \sim \quad \{\phi\}\ \textbf{for}\ \vec{j} \in \mathcal{J}\ \textbf{by}\ \prec_{\mathcal{J}}\ \textbf{do}\ \mathtt{B}(F(\vec{j}))} \end{array}$$

**Fig. 3.** Rule INV-PERMUTE for reordering loop transformations

It is important that the invariant $\phi$ hold at the beginning of the loop and continue to hold (i.e. be invariant) during the execution of the loop. We also require that $\phi$ does not contain any loop index variables, otherwise it may become invalid by the updating of loop index variables at the end of each iteration. Fig. 3 gives the improved INV-PERMUTE rule, which includes an invariant $\phi$ assumed to not contain any reference to the loop index variables.

In INV-PERMUTE, premises 1 and 2 ensure that the permutation $F$ is a bijection, premise 3 ensures that the property $\phi$ holds at the beginning and end of each iteration of the loop, and premise 4 ensures the equivalence of the source and target loop by commutativity. The PERMUTE rule can be regarded as a weaker version of the INV-PERMUTE rule with invariant $\phi = \mathit{true}$.

The following lemma directly implies the soundness of the INV-PERMUTE rule:

**Lemma 1 (Soundess of** INV-PERMUTE**).** *Let $\mathcal{I}$ and $\mathcal{J}$ be finite sets ordered by $\prec_{\mathcal{I}}$ and $\prec_{\mathcal{J}}$ respectively such that $|\mathcal{I}| = |\mathcal{J}|$. Let $F\colon \mathcal{J} \mapsto \mathcal{I}$ be a bijection. Let $\phi$ be a property independent of the loop index variables. If*

$$\forall \vec{i} \in \mathcal{I}: \quad \{\phi\} \quad \mathtt{B}(\vec{i}) \quad \{\phi\}$$

*and*

$$\forall \vec{i}_1, \vec{i}_2 \in \mathcal{I}: \ \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \wedge F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1) \quad \Longrightarrow \quad \{\phi\}\ \mathtt{B}(\vec{i}_1);\mathtt{B}(\vec{i}_2) \ \sim \ \{\phi\}\ \mathtt{B}(\vec{i}_2);\mathtt{B}(\vec{i}_1)$$

*then*

$$\{\phi\}\ \textbf{for}\ \vec{i} \in \mathcal{I}\ \textbf{by}\ \prec_{\mathcal{I}}\ \textbf{do}\ \mathtt{B}(\vec{i}) \quad \sim \quad \{\phi\}\ \textbf{for}\ \vec{j} \in \mathcal{J}\ \textbf{by}\ \prec_{\mathcal{J}}\ \textbf{do}\ \mathtt{B}(F(\vec{j}))$$

*Proof.* Assume that $|\mathcal{I}| = m$, and that $\mathcal{I} = \{\vec{i}_1, \ldots, \vec{i}_m\}$ such that $\vec{i}_1 \prec_{\mathcal{I}} \ldots \prec_{\mathcal{I}} \vec{i}_m$. For every $k = 1, \ldots, m$, let $\mathcal{I}_k = \{\vec{i}_1, \ldots, \vec{i}_k\}$, and denote $\mathcal{J}_k = F^{-1}(\mathcal{I}_k)$. We prove, by induction on $k$, that for all $k = 1, \ldots, m$, if

$$\forall \vec{i}_1, \vec{i}_2 \in \mathcal{I}_k : \ \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \wedge F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1) \quad \Longrightarrow \quad \{\phi\}\ \mathtt{B}(\vec{i}_1); \mathtt{B}(\vec{i}_2) \ \sim \ \{\phi\}\ \mathtt{B}(\vec{i}_2); \mathtt{B}(\vec{i}_1)$$

then

$$\{\phi\}\ \textbf{for}\ \vec{i} \in \mathcal{I}_k\ \textbf{by}\ \prec_{\mathcal{I}}\ \textbf{do}\ \mathtt{B}(\vec{i}) \quad \sim \quad \{\phi\}\ \textbf{for}\ \vec{j} \in \mathcal{J}_k\ \textbf{by}\ \prec_{\mathcal{J}}\ \textbf{do}\ \mathtt{B}(F(\vec{j}))$$

The base case is when $k = 1$ and then the claim is trivial. Assume the claim holds for $k < m$. Denote $F^{-1}(\vec{i}_{k+1})$ by $\vec{j}_*$.

From the induction hypothesis and the properties of $\sim$, it follows that

$$\{\phi\}\ \textbf{for}\ \vec{i} \in \mathcal{I}_{k+1}\ \textbf{by}\ \prec_{\mathcal{I}}\ \textbf{do}\ \mathtt{B}(\vec{i}) \quad \sim \quad \{\phi\}\ \textbf{for}\ \vec{j} \in \mathcal{J}_k\ \textbf{by}\ \prec_{\mathcal{J}}\ \textbf{do}\ \mathtt{B}(F(\vec{j}));\ \mathtt{B}(F(\vec{j}_*))$$

Assume that $\mathcal{J}_k = \{\vec{j}_1, \ldots, \vec{j}_k\}$ such that $\vec{j}_1 \prec_{\mathcal{J}} \ldots \prec_{\mathcal{J}} \vec{j}_k$. If $\vec{j}_* \succ_{\mathcal{J}} \vec{j}_k$, then the inductive step is established. Otherwise, let $\ell$ be the minimal index such that $\vec{j}_* \prec_{\mathcal{J}} \vec{j}_\ell$. It suffices to show that

$$\{\phi\}\mathtt{B}(F(\vec{j}_1)); \ldots; \mathtt{B}(F(\vec{j}_{\ell-1})); \mathtt{B}(F(\vec{j}_*)); \mathtt{B}(F(\vec{j}_\ell)); \ldots; \mathtt{B}(F(\vec{j}_k))$$

$$\sim$$

$$\{\phi\}\textbf{for}\ \vec{j} \in \mathcal{J}_k\ \textbf{by}\ \prec_{\mathcal{J}}\ \textbf{do}\ \mathtt{B}(F(\vec{j}));\ \mathtt{B}(F(\vec{j}_*))$$

Notice that the first assumption

$$\forall \vec{i} \in \mathcal{I} : \quad \{\phi\} \quad \mathtt{B}(\vec{i}) \quad \{\phi\}$$

implies that $\phi$ holds at the beginning and the end of each iteration if $\phi$ holds as precondition of the loop, no matter what the iteration order is. That means:

$$\{\phi\}\ \mathtt{B}(F(\vec{j}_1)); \{\phi\}\ \ldots; \{\phi\}\ \mathtt{B}(F(\vec{j}_{\ell-1})); \{\phi\}\ \mathtt{B}(F(\vec{j}_*));\ \{\phi\}\ \mathtt{B}(F(\vec{j}_\ell)); \{\phi\}\ \ldots; \{\phi\}\ \mathtt{B}(F(\vec{j}_k))\{\phi\}$$

Now, for each $t \in [\ell, \ldots, k]$, we have that $F(\vec{j}_t) \prec_{\mathcal{I}} F(\vec{j}_*)$ and $\vec{j}_* \prec_{\mathcal{J}} \vec{j}_t$, so by R4 of Rule INV-PERMUTE, it follows that

$$\{\phi\}\ \mathtt{B}(F(\vec{j}_t)); \mathtt{B}(F(\vec{j}_*)) \sim \{\phi\}\ \mathtt{B}(F(\vec{j}_*)); \mathtt{B}(F(\vec{j}_t)),$$

and thus $\mathtt{B}(F(\vec{j}_*))$ can be "bubbled" into its position between $\mathtt{B}(F(\vec{j}_{\ell-1}))$ and $\mathtt{B}(F(\vec{j}_\ell))$. □

**Example** Let $\phi$ be the property $k \geq 0$. For the example in Fig. 2, let the the loop index vector $\vec{i_1}$ be the tuple $(i_1, j_1)$, and $\vec{i_2}$ the tuple $(i_2, j_2)$. The domain $\mathcal{I}$ is $[1, N] \times [1, M]$, the domain $\mathcal{J}$ is $[1, M] \times [1, N]$, the permutation function $F$ is $F((j, i)) = (i, j)$, and the body $B((i, j))$ is $A[i + k, j + 1] = A[i, j] + 1;$. The INV-PERMUTE rule requires:

$$\forall i_1, i_2 \in [1, N], \forall j_1, j_2 \in [1, M] : (i_1, j_1) <_{lex} (i_2, j_2) \ \wedge \ (j_2, i_2) <_{lex} (j_1, i_1)$$

$$\Longrightarrow$$

$$\{k \geq 0\} \quad A[i_1 + k, j_1 + 1] = A[i_1, j_1] + 1; \quad A[i_2 + k, j_2 + 1] = A[i_2, j_2] + 1;$$

$$\sim$$

$$\{k \geq 0\} \quad A[i_2 + k, j_2 + 1] = A[i_2, j_2] + 1; \quad A[i_1 + k, j_1 + 1] = A[i_1, j_1] + 1;$$

Let $read(A, i)$ denote the value obtained by "reading" the $i$th element of array $A$, and $write(A, i, x)$ denote a new array obtained by "writing" $x$ to the $i$th element of array $A$. The above verification condition can then be expressed as:

$$1 \le i_1 \le N \wedge 1 \le i_2 \le N \wedge 1 \le j_1 \le M \wedge 1 \le j_2 \le M \wedge\ i_1 < i_2 \wedge j_1 > j_2$$

$$\Longrightarrow$$

$$k \ge 0 \Longrightarrow$$

$$\begin{array}{c} (A_1 = write(A, (i_1 + k, j_1 + 1), read(A, (i_1, j_1)) + 1) \\ \wedge\ A_2 = write(A_1, (i_2 + k, j_2 + 1), read(A_1, (i_2, j_2)) + 1) \\ \wedge\ A'_1 = write(A, (i_2 + k, j_2 + 1), read(A, (i_2, j_2)) + 1) \\ \wedge\ A'_2 = write(A'_1, (i_1 + k, j_1 + 1), read(A'_1, (i_1, j_1)) + 1)) \end{array}$$

$$\Longrightarrow$$

$$A_2 = A'_2$$

which can be verified as a valid formula by an automated theorem prover.

The new permutation rule INV-PERMUTE can be used by a compiler to decide whether some loop transformation is valid at compile time given a loop invariant determined by static analysis. Because an appropriate invariant is generally hard to find, we use an automatic theorem prover, CVC Lite [3], to try to generate a condition under which the loop transformation is valid. This condition can then be checked in the loop to see whether it is indeed invariant. This paper will not provide the algorithm for generating such a condition using CVC Lite. You may refer to [10] for the details of the algorithm.

## 4 Rule Reduce

| | | |
|---|---|---|
| **for** $i = 1$ **to** $N$ **do**<br>$x := x + 1;$ | $\Longrightarrow$ | $x := x + N;$ |

**Fig. 4.** An example for loop reduction.

$$\begin{array}{lll} \text{R1.} & & B(1) \sim B'(1) \\ \text{R2.} & \forall i > 0: & B'(i); B(i+1) \sim B'(i+1) \\ \hline & & \textbf{for } i = 1 \textbf{ to } N \textbf{ do } B(i) \sim B'(N) \end{array}$$

**Fig. 5.** Rule REDUCE for loop reduction.

Rule INV-PERMUTE can handle any loop reordering transformation, but there are other kinds of loop transformations that cannot be handled by either VALIDATE or INV-PERMUTE. Fig. 4 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. 5, where the symbol $\sim$ means that two pieces of code are equivalent.

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

**Fig. 6.** Reduction for an empty loop.

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. 4, $B(i)$ 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. 6 shows a transformation which removes a loop with no loop body. In this case, $B(i) = B'(i) =$ **Skip**.

```
CP1 :
   for i = 1 to N do
CP2 :
      for j = 1 to M do
CP3 :
         B(i, j);
CP4 :
```

$\Longrightarrow$

```
cp1 :
   if (1 ≤ N) then {
l1 :
      if (1 ≤ M) then {
         for i = 1 to N do
cp2 :
            for j = 1 to M do
cp3 :
               B(i, j);
      }
   }
cp4 :
```

**Fig. 7.** An example for which rule VALIDATE fails.

## 5 A Generalization of Rule Validate

Section 2 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. 7 gives an example of such a transformation performed by ORC.

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. 7 is that it assumes each simple path in the target corresponds to one or more simple paths in the source. However, this transformation transforms

0. Establish source and target cut-point sets $\mathsf{CP}_S$ and $\mathsf{CP}_T$, which include all initial and terminal program locations. For any simple path between two cut-points $i$ and $j$, its transition relation $\rho_{ij}$ must be computable.
1. Establish a *control abstraction* $\kappa$: $\mathsf{CP}_T \to \mathsf{CP}_S$ 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 $\mathsf{CP}_T$, form an *invariant* $\varphi_i$ 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

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

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 $\pi$.
5. Establish the validity of all the generated verification conditions.

**Fig. 8.** The generalized rule GEN-VALIDATE

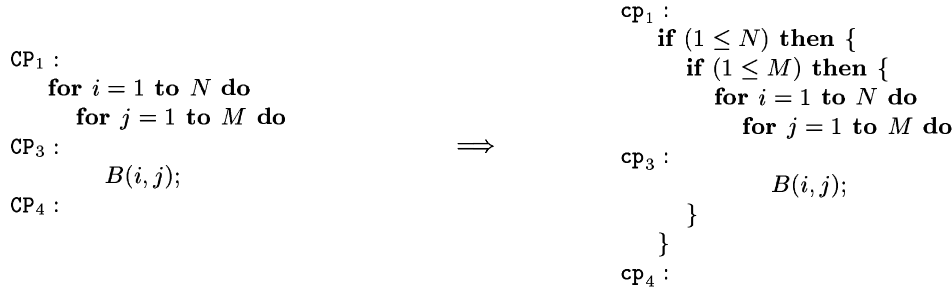


**Fig. 9.** Example with modified cut-points.

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. 8, and a proof of its correctness is given in the appendix. It is essentially the same proof rule as that given in [23] 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. 7, we can eliminate cut-points $\mathtt{CP}_2$ and $\mathtt{cp}_2$ as shown in Fig. 9. 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. 6), 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 $1 \le M$ 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 $\mathtt{cp}_4$.

The proof of the soundess of rule GEN-VALIDATE is provided as follows.

**Soundess of** GEN-VALIDATE

*Proof.* 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 $t_0$, $\sigma_T$ is terminating (finite) iff $\sigma_S$ 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 $\sigma_T$. 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,

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

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 $\sigma_S$. Now, suppose the corresponding target computation $\sigma_T$ 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 $\sigma'_S$ with an infinite number of source cut-points. This would require there to be two different source computations $\sigma_S$ and $\sigma'_S$ starting from the same initial source state $s_0$, which violates the assumption that the source program is deterministic. Therefore, the corresponding target computation $\sigma_T$ must be terminating. And according to the previous argument, their final states must be compatible.

## 6 Conclusion

This paper describes enhancements to our translation validation framework, primarily allowing additional loop and loop-related transformations to be validated. We introduced the new proof rule INV-PERMUTE for reordering loop optimizations considering the initial and invariant conditions of loops, rule REDUCE for loop reduction, and a generalization of the old validate rule GEN-VALIDATE.

We have implemented an automatic validation tool called TVOC according to the theory in this paper, and succeeded in handling many examples with loop transformations. We plan to continue to improve the theory and implementation of translation validation for optimizing compilers. In particular, we hope to implement more loop transformations and to try our tool on larger examples.

## References


1. A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1988.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.
3. 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.
4. C. Barrett, B. Goldberg, and L. Zuck. Run-time validation of speculative optimizations using CVC. In O. Sokolsky and M. Viswanathan, editors, *Third International Workshop on Run-time Verification (RV)*, pages 87–105, July 2003. Boulder, Colorado, USA.
5. 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.
6. R. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19:19–32, 1967.
7. 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.
8. G. Goos and W. Zimmermann. Verification of compilers. *Lect. Notes in Comp. Sci.*, 1710:201–230, 1999.
9. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
10. Ying Hu, Clark Barrett, and Benjamin Goldberg. Theory and algorithms for the generation and validation of speculative loop optimizations. In *2nd IEEE International Conference on Software Engineering and Formal Methods*, 2004.
11. C. Jaramillo, R. Gupta, and M. Soffa. Debugging and testing optimizers through comparison checking. *Lect. Notes in Comp. Sci.*, 65(2), 2002.
12. R.-C. Ju, S. Chan, and C. Wu. Open research compiler (orc) for the itanium processor family. In *Micro 34*, 2001.
13. 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.
14. Y. Hu, C. Barrett, B. Goldberg, and L. Zuck. TVOC: A tool for the translation validation of optimizing compilers. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, Apr. 2004.
15. G. Necula. Proof-carrying code. In *POPL'97*, pages 106–119, 1997.
16. 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.
17. 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.
18. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*, pages 151–166, 1998.
19. M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, 1999.
20. M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. of the SIGPLAN '91 Symp. on Programming Language Design and Implementation*, pages 33–44, 1991.
21. M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.
22. 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.
23. 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.