Translation Validation of Loop Optimizations

by

Ying Hu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 2005

Benjamin Goldberg

Clark Barrett

ACKNOWLEDGEMENTS

I would like to thank my advisers Benjamin Goldberg and Clark Barrett for their support and guidance over many years. I would also like to thank Amir Pnueli and Lenore Zuck for their valuable advice. Many others have contributed to the success of this work, including my family and friends.

ABSTRACT

This dissertation presents a translation validation framework for verifying optimizing compilers. Our validation work is focused on loop optimizations since loops are generally hard to analyze and it is especially difficult to verify loop optimizations. To solve these problems, a set of theories and algorithms are proposed, and they are implemented in our validation tool called TVOC.

In order to validate the optimizations performed by the compiler, we try to prove the equivalence of intermediate representaions of a program before and after the optimizations. A set of proof rules were previously proposed to build the equivalence relation between two programs. However, they cannot validate some cases with legal loop optimizations. We improve these proof rules to consider the conditions of loops and possible elimination of some loops, so that those cases can also be handled. Algorithms are designed to apply the new rules to an automatic validation process.

Based on the proof rules mentioned above, our validation tool TVOC validates compiler optimizations by analyzing the original and optimized codes and generating checkable verification conditions. Previously, TVOC only dealt with optimizations which do not significantly change the structure of the code, while loop optimizations do change the structure greatly. We built a new part of TVOC treating loop optimizations seperately, which guesses what kinds of loop optimizations happened, analyzes the loops, proves the validity of a com-

bination of loop optimizations, and synthesizes a series of intermediate codes. With this new component, TVOC has succeeded in validating many examples with loop optimizations.

Speculative optimizations are aggressive optimizations which are only correct under certain conditions that cannot be known at compile time. In this dissertation, we present a formalism for validating speculative optimizations and generating the runtime tests necessary to ensure their correctness. We also provide results on several examples.

Contents

List of Figures

Chapter 1

Introduction

There is a growing awareness in industry and academia of the crucial role of formally proving the correctness of systems. For a critical software system, 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 the 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. The primary goal of my research has been integrating an approach for validating loop transformations into a Translation Validation framework for optimizing compilers. An automatic verification tool, called the Translation Validator for Optimizing Compilers (TVOC), has been designed and implemented. This thesis discusses the theory

and the system integration and implementation issues relating to validation of loop transformations.

Traditional compiler verification treats a compiler as a program, and tries to prove the correctness of the algorithms and their implementation, which becomes a difficult task due to the complexity and evolution of modern compilers. Recently, a number of creative techniques for verifying compilers have been introduced [Nec97, NL98, RM00, GS99, JGS02, PSS98b, Nec00, LMC03].

In [Nec97] [NL98], 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.

[Nec00] verifies the preservation of semantics for each compilation which is the same as ours. 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* [RM00] 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 [Nec00].

In [GS99], 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 [JGS02] 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 runtime of the program.

In [LMC03], compiler optimizations are automatically proved correct using the automatic theorem prover Simplify [DNS03]. 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, and it does not handle loop optimizations.

Our approach, translation validation [PSS98b], 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, it can be used to verify existing compilers due to its independence from the compiler, and it does not increase the runtime of programs.

Previous work on TV focused on validating optimizations which preserve

the loop structure of the code, while more aggressive optimizations, especially loop optimizations were not considered. A set of proof rules and algorithms were proposed to check the validity of loop transformations by treating loop transformations as permutations. I have successfully implemented these permutation rules to check the validity of a number of reordering loop transformations such as loop fusion/distribution, loop interchange and tiling. However, these proof rules have the limitation of requiring the loop transformations to be valid in all contexts without considering any conditions outside of the loop. Therefore, I proposed an improved proof rule to consider the initial conditions and invariant conditions of the loop. This new proof rule not only improves the validation process for compile-time optimizations, it can also be used to ensure the correctness of speculative loop optimizations, aggressive optimizations which are only correct under certain conditions that cannot be known at compile time. Based on the new rule, with the help of an automatic theorem prover, CVC Lite, an algorithm was presented for validating loop optimizations. The improved proof rule can also be used to generate the runtime tests necessary to support speculative optimizations.

A previous proof rule VALIDATE was used to validate structure-preserving transformations with a clear mapping of control points between target and source programs. However, this rule was found 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 transformations, I introduced a new proof rule Reduce to break a loop down to a block of non-loop statements. In addition, I observed that in structure-preserving cases involving nested loops, rule VALIDATE was unsuccessful. Therefore rule VALIDATE was generalized to have less restrictive requirements, allowing the set of control points to be chosen more freely.

Finally, based on the above theory and algorithms, I contributed in developing TVOC, a tool for the translation validation of advanced optimizing compilers. In the framework of TV, to prove that the translated code correctly implements the original code, TVOC uses the proof rule VALIDATE for structure preserving optimizations, and the permutation rule for loop optimizations. TVOC accepts as input intermediate codes produced by the compiler. Just as compilers perform optimizations in multiple passes, TVOC breaks the validation into multiple phases, each using a different proof rule and focusing on a different set of optimizations. In each phase, TVOC automatically generates verification conditions and proves their validity using CVC Lite.

This thesis is organized as follows: Chapter 1 gives the background for the translation validation framework, loop optimizations and proof rules VALIDATE and PERMUTE. Chapter 2 explains the theoretical work I have done for translation validation. Chapter 3 introduces the tool TVOC. Chapter 4

discusses the theory and implementation for speculative loop optimizations. Chapter 5 concludes.

Chapter 2

Background

This chapter gives the background for my thesis work. Section 2.1 describes Transition Systems. Section 2.2 introduces the notation of Translation Validation. Section 2.3 reviews a set of general loop optimizations. Section 2.4 and Section 2.5 present the proof rules PERMUTE and VALIDATE.

2.1 Transition Systems

In order to discuss the formal semantics of programs, we briefly review *transition systems*, TS's, a variant of the *transition systems* of [PSS98b]. A *Transition System* $S = \langle V, \mathcal{O}, \Theta, \rho \rangle$ is a state machine consisting of: 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 unprimed versions of variables refer to the values of the variables in the pre-transition state, while the primed versions of variables refer to their values in the successor states. Thus, e.g., the transition relation may include "$x' = x + 1$" to denote that the value of the variable x 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:

L_0:

$n := 100$;

if $(n < x)$ **goto** L_2;

L_1:

...

L_2:

...

There are two disjuncts in the transition relation whose starting locations are L_0. The first describes the L_0 to L_1 path, which is $pc = L_0 \ \wedge \ n' = 100 \ \wedge \ x' = x \ \wedge \ n' \geq x' \ \wedge \ pc' = L_1$, and the second describes the L_0 to L_2 path, which is $pc = L_0 \ \wedge \ n' = 100 \ \wedge \ x' = x \ \wedge \ n' < x' \ \wedge \ pc' = L_2$. 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/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 $\sigma : s_0, s_1, \ldots$, where the starting state s_0 satisfies the initial condition Θ, and every two consecutive states are related by the transition relation ρ.

A transition system $\mathcal{T}$ can be deterministic or non-deterministic. If the observable part of the initial condition uniquely determines the rest of the computation, the transition system is called *deterministic* . This thesis is focused on 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.

We denote $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ as the source TS and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ as the target TS. If there exists a one-to-one correspondence between the observables of P_S and those of P_T, the two systems are called *comparable* . 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 between *structure preserving* optimizations, with 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, while most loop optimizations are structure modifying.[1]

2.2 Translation Validation

A compiler is defined [ASU86] as a program that reads a program written in the source language, and translates it into an equivalent program in the target language (Fig. 2.1).

Figure 2.1: A compiler.

Correctness of compilers was first considered by McCarthy in [MP67], where a simple compilation of arithmetic expressions was verified manually.

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

Traditional compiler verification treats a compiler as a program, and tries to prove the correctness of the algorithms and their implementations, which becomes a difficult task due to the complexity and frequent evolution of modern compilers.

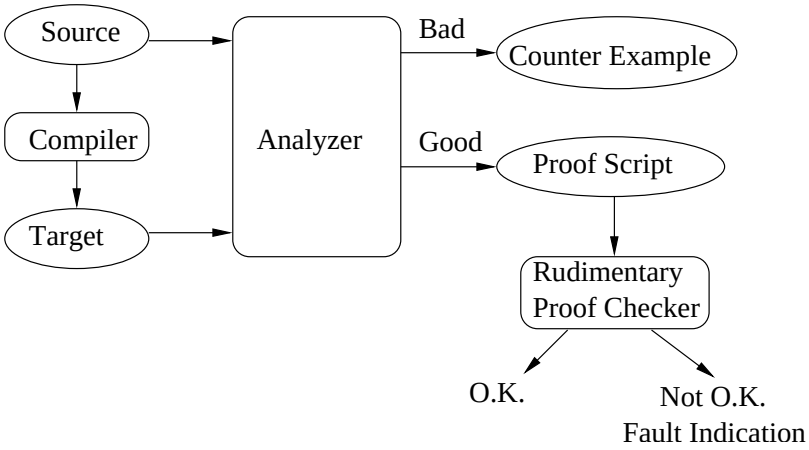

Figure 2.2: Translation Validation.

Pnueli et al [PSS98b] proposed the notion of translation validation, which proves the target program preserves the meaning of the source program by analyzing the results of each run of the compiler instead of analyzing the compiling process.

Fig. 2.2 taken from figure 1 in [PSS98b] gives an overview of translation validation, where the source and target programs are sent to an *analyzer* which compares them and produces a proof script if the target program correctly implements the source program, or emits a counter-example otherwise. The proof

script is a witness of the correct compilation and it will be further checked by a proof checker, while the counter-example is an evidence of bad compilation.

The definitions in [PSS98b] built a framework for translation validation, proving the target program implements the source program inductively by a mapping of observable states. They are the theoretical basis for the later work in [ZPFG03], which refined the innovative concepts in [PSS98b] for establishing the equivalence of two transition systems, using control points of programs instead of observable states and introducing a data mapping instead of assuming the observable set of the target is a subset of that of the source. Inspired by Floyd's inductive method, using the transition system model instead of the flow chart, they proposed a new proof rule VALIDATE which inductively proves that a target program correctly implements a source program by establishing an equivalence relation for each pair of corresponding source and target control points. Without help from the compiler, the control points of a target program were mapped in order with the control points in a source program, thus this rule requires the translation to be *structure preserving* with a clear mapping of control points between target and source programs.

2.3 Loop Optimizations

Loop transformations attempt to improve performance by rewriting loops to make better use of the memory hierarchy or increase parallelism. This section gives definitions of loop transformations including loop fusion, distribution, peeling, interchange, tiling, unrolling, reversal, skewing, and scalar replacement according to [AK02].

1. Loop fusion and distribution

Loop fusion merges two or more loops into a single loop. It is often used to avoid loading those memory locations into the cache multiple times and to enhance temporal locality.

Here is an example of loop fusion:

```
Do I = 1 to N                          Do I = 1 to N
  A[I] = B[I] + 1;                       A[I] = B[I] + 1;
                          ⟹
Do I = 1 to N                            D[I] = A[I] + C;
  D[I] = A[I] + C;
```

When transformed from right to left, the above is an example for loop distribution, which splits a single loop into several separate loops. Loop distribution is simetimes used to reduce the amount of memory used during one

loop so that the remaining memory may fit in the cache, and it can be used to convert a sequential loop to multiple parallel loops.

2. Loop peeling

Loop peeling moves the first iteration of a loop outside the loop. Peeling a loop may expose the code to other optimizations. For example:

```
Do I = 1 to N                          A[1] = A[1] + A[1];
  A[I] = A[I] + A[1]; ⟹                Do I = 2 to N
                                         A[I] = A[I] + A[1];
```

The resulting loop can be vectorized:

```
A[1] = A[1] + A[1];

A[2:N] = A[2:N] + A[1];
```

3. Loop interchange

Loop interchange changes the nesting order of loops in a perfect nest. It can minimize the stride of array element access during loop execution and reduce the number of memory accesses needed. For example:

```
Do I = 1 to N                          Do J = 1 to M
 Do J = 1 to M                          Do I = 1 to N
  A[I,J+1] = A[I,J]+B;        ⟹          A[I,J+1] = A[I,J]+B;
```

after loop interchange the inner loop can vectorize to produce:

```
Do J = 1 to M
  A[1:N,J+1] = A[1:N,J] + B;
```

In this example, loop interchange enhances vectorization by moving a vectorizable loop to the innermost position.

4. Loop tiling

Loop tiling decomposes an n-dimensional loop nest into a 2n-dimensional loop nest, where the inner n-loops together scan the iterations in a tile of the original iteration space. It can improve the cache performance. A standard example is matrix multiplication:

```
Do i = 1 to n
  Do j = 1 to n
    Do k = 1 to n
      c[i,j]=c[i,j]+a[i,k]*b[k,j];
```

⟹

```
Do i0 = 1 to n step b
  Do j0 = 1 to n step b
    Do k0 = 1 to n step b
      Do i = i0 to min(i0+b-1,n)
        Do j = j0 to min(j0+b-1,n)
          Do k = k0 to min(k0+b-1,n)
            c[i,j]=c[i,j]+a[i,k]*b[k,j];
```

5. Loop unrolling

Loop unrolling repeats loop body instructions several times within a single loop iteration. It minimizes the number of branches and groups more instructions together to allow efficient overlapped instruction execution. For example:

```
Do I = 1 to N
  A[I] = 0
```

$\Longrightarrow$

```
Do I = 1 to N by 4
  A[I] = 0;
  A[I+1] = 0;
  A[I+2] = 0;
  A[I+3] = 0;
```

6. Loop reversal

Loop reversal runs a loop backward. It provides an opportunity for other transformations. For example:

```
Do I = 1 to N                          Do I = N to 1 by -1
  A[I] = B[I] + 1;                       A[I] = B[I] + 1;
  C[I] = A[I]/2;              ⟹         C[I] = A[I]/2;
Do I = 1 to N                          Do I = N to 1 by -1
  D[I] = 1/C[I+1]                        D[I] = 1/C[I+1]
```

After reversal, the loop allows fusion:

```
Do I = N to 1 by -1
  A[I] = B[I] + 1;
  C[I] = A[I]/2;
  D[I] = 1/C[I+1]
```

7. Loop skewing

Loop skewing reshapes an iteration space to allow parallelization. For example:

```
Do I = 1 to N
  Do J = 1 to M
    A[I,J]=A[I-1,J]+A[I,J-1];
```

$\Longrightarrow$

```
Do I = 1 to N
  Do j = I+1 to I+M
    A[I,j-I]=A[I-1,j-I]+A[I,j-I-1];
```

The index j is used to replace index J with $J = j - I$. After skewing, the loop can be interchanged into:

```
Do j = 2 to M+N
  Do I = max(1,j-N) to min(N,j-1)
    A[I,j-I]=A[I-1,j-I]+A[I,j-I-1];
```

After loop skewing and interchange, the inner loop can be vectorized.

8. Scalar replacement

Scalar replacement replaces the use of an array element with a scalar variable. It reduces memory references. For example:

```
Do I = 1 to N                          Do I = 1 to N
                                          t = A[I];
  Do J = 1 to M                           Do J = 1 to M
                          ⟹
    A[I] = A[I]+B[J];                        t = t+B[J];
                                          A[I] = t;
```

Here scalar replacement exposes the reuse of A[I].

2.4 Rule PERMUTE

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.

$$\begin{array}{l} \textbf{for } i_1 = L_1, H_1 \textbf{ do} \\ \quad \ldots \\ \qquad \textbf{for } i_m = L_m, H_m \textbf{ do} \\ \qquad\quad B(i_1, \ldots, i_m) \end{array}$$

Figure 2.3: A General Loop

Consider the generic loop in Fig. 2.3.

Schematically, we can describe such a loop as "**for** $\vec{i} \in \mathcal{I}$ **by** $\prec_{\mathcal{I}}$ **do** $\mathtt{B}(\vec{i})$" where $\vec{i} = (i_1, \ldots, i_m)$ is the vector of nested loop indices, and $\mathcal{I}$ is the set of the values assumed by $\vec{i}$ through the different iterations of the loop. The set $\mathcal{I}$ can be characterized by a set of linear inequalities. For example, for the loop of Fig. 2.3,

$$\mathcal{I} = \{(i_1, \ldots, i_m) \mid L_1 \leq i_1 \leq H_1 \ \wedge \ \cdots \ \wedge \ L_m \leq i_m \leq H_m\}.$$

The relation $\prec_{\mathcal{I}}$ is the ordering by which the various points of $\mathcal{I}$ are traversed. For example, for the loop of Fig. 2.3, this ordering is the lexicographic order on $\mathcal{I}$.

Consider a generic loop transformation:

$$\textbf{for } \vec{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\vec{i})$$

$$\Longrightarrow$$

$$\textbf{for } \vec{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\vec{j}))$$

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 $\prec_{\mathcal{J}}$, the permutation function F is a mapping from $\mathcal{J}$ to $\mathcal{I}$, and the loop body B is parameterized by the loop index vector.

In [ZPFG03], rule PERMUTE was proposed for validating such loop reordering transformations. As shown in Fig. 2.4, 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. The symbol $\sim$ in Fig. 2.4 means that two pieces of code are equivalent, i.e. they transform the program state in the same way.

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

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

Figure 2.4: Rule PERMUTE for Reordering Transformations

As shown, rule PERMUTE may appear to deal only with transformations that change the execution order of a single loop body. However, as shown in [GZB04], by considering more sophisticated domains for $\mathcal{I}$, it is possible to handle a wide variety of loop structures including multiple and nested loops. Rule PERMUTE is limited to reordering transformations that are valid in all contexts. In Section 3.1, we will introduce a more general proof rule INV-PERMUTE [HBG04] which is able to consider the initial and invariant condi-

tions of the loop.

2.5 Rule VALIDATE

In the translation validation framework, the validator compares the source and target programs, and tries to verify their equivalence. To prove the equivalence of two programs with the same structure, we use a proof rule, VALIDATE (see [ZPFG03], and a variant in [ZPG+05] which produces simpler verification conditions), which is inspired by the computational induction approach ([Flo67]), originally introduced for proving properties of a single program.

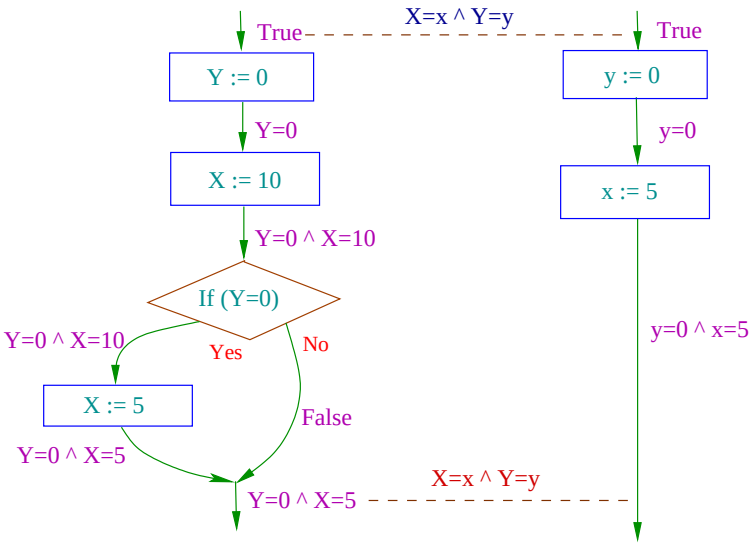

Figure 2.5: Program equivalence with no loop

Fig. 2.5 gives the flow charts for a simple source code (on the left) and

its target code (on the right). In a flow chart, each vertex (such as Y:=0) is a statement, and the edges tagged by assertions (such as Y=0) are the possible control passages between the statements. For each vertex, the proof task is that the properties of its antecedent edges implies the properties of its consequent edges. The properties at the final edge of a flow chart can be proved inductively given the properties at the initial edge of the flow chart. It is obvious that $Y = 0 \ \wedge \ X = 5$ holds at the final edge of the source flow chart, and $y = 0 \ \wedge \ x = 5$ holds at the final edge of the target flow chart.

This inductive method for proving the properties of *one* program can be extended to prove the equivalence of *two* programs, if the initial and terminal points of the target is mapped into the initial and terminal points of the source, and the observable variables of the two programs are mapped (such as X corresponding to x, Y corresponding to y). This equivalence usually requires that the observable variables have the same values (such as $X = x \wedge Y = y$) at the final points of both programs, given the same initial inputs. This seems to be straightforward for the simple example in Fig. 2.5, since it is easy to get the relations between the initial states and final states of the programs. However, for the cases with loops (Fig. 2.6), it can be hard to get the relation bewteen the initial state and final state of a program. Our strategy is to introduce selected control points to break the cycles: besides the initial and terminal points, there is at least one point for each cycle. When the target control

point set is mapped into the source control point set, given an equivalence relation at a pair of corresponding control points, it can be proved that the relation also holds for the next pairs of corresponding control points. Thus it can be proved inductively that the equivalence relation holds at the final pairs of control points of the programs.

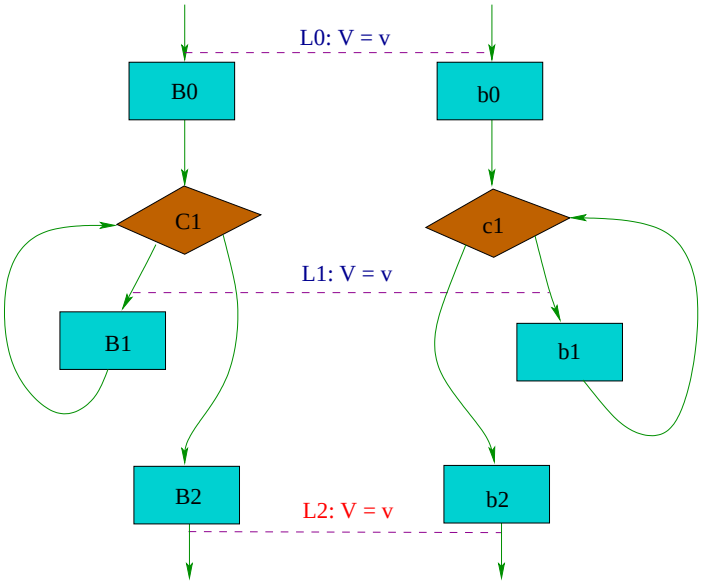

Figure 2.6: Program equivalence with loops

With the above intuition, assuming both source and target programs are deterministic and have the same inputs, rule VALIDATE in Fig. 2.7 (this version of rule VALIDATE is from [ZPG+05] paper, and it differs slightly from its previous version in [ZPFG03] paper) provides a proof methodology by which one can prove that one program *refines* another. This is achieved by estab-

1. Establish a *control abstraction* κ: $\mathsf{CP}_T \rightarrow \mathsf{CP}_S$ such that i is an initial block of T iff $\kappa(i)$ is an initial block of S and i is a terminal block of T iff $\kappa(i)$ is a terminal block of S.

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

3. Establish a *data abstraction*

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

   which asserts that the source and target are at corresponding blocks and which assigns to *some* non-control source state variables $V_i \in V_S$ an expression e_i over the target state variables, conditional on the (target) boolean expression p_i. Note that α may contain more than one clause for the same variable. It is required that for every initial target block `Bi`, $\Theta_S \ \wedge \ \Theta_T \rightarrow \alpha \ \wedge \ \varphi_i$. It is also required that for every *observable* source variable $V \in \mathcal{O}_S$ (whose target counterpart is v) and every terminal target block `B`, α implies that $V = v$ at `B.`

4. For each pair of basic blocks `Bi` and `Bj` such that there is a simple path from `Bi` to `Bj` in the control graph of P_T, we form the verification condition

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

   where $Paths(\kappa(i))$ is the set of all simple source paths from $\kappa(i)$ 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.

Figure 2.7: The Proof Rule VALIDATE

lishing 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 rule VALIDATE, $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ are two comparable TS's, where P_S is the *source* and P_T is the *target*. Each TS is assumed to have a *cut-point* set, a subset of the program locations (i.e. possible values of pc) that includes all initial and terminal locations, as well as at least one location from each of the cycles in the programs' 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. Typically, such a transition relation contains the condition 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.

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

Chapter 3

Validating Loop Optimizations

This chapter decribes the theoretical work I have done for my thesis. Section 3.1 explains rule INV-PERMUTE. Section 3.2 gives rule REDUCE. Section 3.3 presents rule GEN-VALIDATE. Section 3.4 concludes.

3.1 Improved PERMUTE Rule

A modern compiler performs a set of advanced optimizations to make the compiled code run faster. Among them are loop optimizations which improve parallelism and make efficient use of the memory hierarchy. A *reordering transformation* is defined [AK02] as any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statement. Many loop transformations, including rever-

sal, fusion, distribution, interchange, and tiling, are in the class of reordering transformations.

Traditionally, dependence analysis has been used to determine whether it is safe to perform certain kinds of program transformations. In the presence of two accesses to the same memory location (where at least one is a write) dependence theory [AK02] states that a reordering transformation *preserves* a dependence if it preserves the relative execution order of the source and target (i.e. the first memory access and second memory access) of that dependence. A reordering transformation is *valid* if it preserves all dependences in the program. To decide whether a reordering loop transformation preserves the meaning of the program, the compiler usually performs dependence analysis. The basic idea is that for any pair of statements s_1 and s_2, if there is any dependence between them, then the order of executing them cannot be changed in the transformation. Let $Dependence(s_1, s_2)$ be a predicate denoting whether there is any dependence between the statements s_1 and s_2, and let $Reorder(s_1, s_2)$ be a predicate denoting whether the transformation can safely reorder the execution of s_1 and s_2. The dependence rule can be schematized as:

$$Dependence(s_1, s_2) \implies \neg Reorder(s_1, s_2) \tag{3.1}$$

or, equivalently,

$$Reorder(s_1, s_2) \implies \neg Dependence(s_1, s_2) \tag{3.2}$$

This rule has a stronger requirement than necessary for the correctness of reordering loop transformations. The right hand side requires that there is no dependence between the pair of statements, but there are cases when this is too conservative. For example, when two statements assign the same value to a variable, it does not matter which one is executed first. From a broader view of program equivalence, let $s_1; s_2; \sim s_2; s_1;$ denote that the effect of executing the two statements s_1 and s_2 in either order is the same. The rule becomes:

$$Reorder(s_1, s_2) \implies s_1; s_2; \sim s_2; s_1; \tag{3.3}$$

Rule (3.3) is more powerful than rule (3.2), because it validates more cases than rule (3.2). Rule PERMUTE (Fig. 2.4) formalizes the idea in rule (3.3). In addition to being more general, the PERMUTE rule has two additional advantages over the standard dependence analysis approach. First, it only needs the information inside the loop to generate the logical formula for code equivalence, without explicitly having to perform dependence analysis. Second, PERMUTE can leave the task of proving the legality of transformations to an automatic theorem prover, which can not only determine whether a transformation is

legal, but can actually provide a proof in the case that it is[1].

```
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
```

Figure 3.1: A loop interchange example

Though it is easy to implement, 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. 2.4 to be valid for all values of non-index variables. Consider the loop in Fig. 3.1, 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 ϕ (such as $k \geq 0$), so that premise 3 becomes:

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

$$\Longrightarrow$$

$$\{\phi\}\ \mathtt{B}(\vec{i}_1); \mathtt{B}(\vec{i}_2) \ \sim \ \{\phi\}\ \mathtt{B}(\vec{i}_2); \mathtt{B}(\vec{i}_1)$$

where the representation $\{\phi\}$ uses Hoare's precondition notation [Hoa69],

[1] In fairness, much of the machinery required to perform dependence analysis, including solving diophantine equations involving array subscripts, must be incorporated into the theorem prover.

meaning that we assume ϕ holds before each of the two pieces of code.

R1. $\forall \vec{i} \in \mathcal{I} : \exists \vec{j} \in \mathcal{J} :$ $\quad \vec{i} = F(\vec{j})$

R2. $\forall \vec{j}_1 \neq \vec{j}_2 \in \mathcal{J} :$ $\quad F(\vec{j}_1) \neq F(\vec{j}_2)$

R3. $\forall \vec{i} \in \mathcal{I} :$ $\quad \{\phi\} \;\; \mathtt{B}(\vec{i}) \;\; \{\phi\}$

R4. $\forall \vec{i}_1, \vec{i}_2 \in \mathcal{I} :$ $\quad \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \;\wedge\; F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1)$
$\quad \Longrightarrow \quad \{\phi\}\; \mathtt{B}(\vec{i}_1); \mathtt{B}(\vec{i}_2) \;\sim\; \{\phi\}\; \mathtt{B}(\vec{i}_2); \mathtt{B}(\vec{i}_1)$

$\{\phi\}$ **for** $\vec{i} \in \mathcal{I}$ **by** $\prec_{\mathcal{I}}$ **do** $\mathtt{B}(\vec{i})$

$\sim$

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

Figure 3.2: Rule INV-PERMUTE for reordering loop transformations

It is important that the invariant ϕ 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 ϕ 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. 3.2 gives the improved INV-PERMUTE rule [HBG04], which includes an invariant ϕ 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 ϕ 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 $\phi = true$.

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

Lemma 3.1.1 (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 ϕ 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

$$\forall \vec{i}_1, \vec{i}_2 \in \mathcal{I} : \ \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \wedge F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1)$$

$$\Longrightarrow$$

$$\{\phi\}\ \mathtt{B}(\vec{i}_1); \mathtt{B}(\vec{i}_2) \ \sim \ \{\phi\}\ \mathtt{B}(\vec{i}_2); \mathtt{B}(\vec{i}_1)$$

then

$$\{\phi\}\ \textbf{for}\ \vec{i} \in \mathcal{I}\ \textbf{by}\ \prec_{\mathcal{I}}\ \textbf{do}\ \mathtt{B}(\vec{i})$$

$$\sim$$

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

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

$$\forall \vec{i}_1, \vec{i}_2 \in \mathcal{I}_k : \ \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \wedge F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1)$$

$$\Longrightarrow$$

$$\{\phi\}\ \mathtt{B}(\vec{i}_1); \mathtt{B}(\vec{i}_2) \ \sim \ \{\phi\}\ \mathtt{B}(\vec{i}_2); \mathtt{B}(\vec{i}_1)$$

then

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

$$\sim$$

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

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

$$\{\phi\}\ \textbf{for}\ \vec{i} \in \mathcal{I}_{k+1}\ \textbf{by}\ \prec_{\mathcal{I}}\ \textbf{do}\ \mathtt{B}(\vec{i})$$

$$\sim$$

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

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 ℓ be the minimal index such that $\vec{j}_* \prec_{\mathcal{J}} \vec{j}_\ell$. It suffices to show that

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

$$\sim$$

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

Notice that the first assumption

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

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

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

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 ϕ be the property $k \geq 0$. For the example in Fig. 3.1, let 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:

$$\forall i_1, i_2 \in [1, N], \forall j_1, j_2 \in [1, M] :$$

$$(i_1, j_1) <_{lex} (i_2, j_2) \ \wedge \ (j_2, i_2) <_{lex} (j_1, i_1)$$

$$\Longrightarrow$$

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

$$A[i_2 + k, j_2 + 1] = A[i_2, j_2] + 1;$$

$$\sim$$

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

$$A[i_1 + k, j_1 + 1] = A[i_1, j_1] + 1;$$

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

$$1 \leq i_1 \leq N \wedge 1 \leq i_2 \leq N \wedge 1 \leq j_1 \leq M \wedge 1 \leq j_2 \leq M$$

$$\wedge\ i_1 < i_2 \wedge j_1 > j_2$$

$$\Longrightarrow$$

$$k \geq 0$$

$$\Longrightarrow$$

$$\begin{aligned} & (A_1 = write(A, (i_1 + k, j_1 + 1), read(A, (i_1, j_1)) + 1) \\ \wedge \quad & A_2 = write(A_1, (i_2 + k, j_2 + 1), read(A_1, (i_2, j_2)) + 1) \\ \wedge \quad & A_1' = write(A, (i_2 + k, j_2 + 1), read(A, (i_2, j_2)) + 1) \end{aligned}$$

$$\wedge \quad A'_2 = write(A'_1, (i_1 + k, j_1 + 1), read(A'_1, (i_1, j_1)) + 1))$$

$$\Longrightarrow$$

$$A_2 = A'_2$$

which can be verified as a valid formula by the automated theorem prover CVC Lite.

3.2 Rule REDUCE

for $i = 1$ **to** N **do**
  $x := x + 1;$
$\Longrightarrow$
$x := x + N;$

Figure 3.3: 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. 3.3 shows an example (an actual transformation performed by Intel's Open Research Compiler [JCW01]) in which a loop is removed and replaced with a single statement. We call this **loop reduction** and propose a new proof rule, REDUCE [HBGP05], to deal with such cases. Rule REDUCE is shown in Fig. 3.4, where the symbol $\sim$ means that two pieces of code are equivalent.

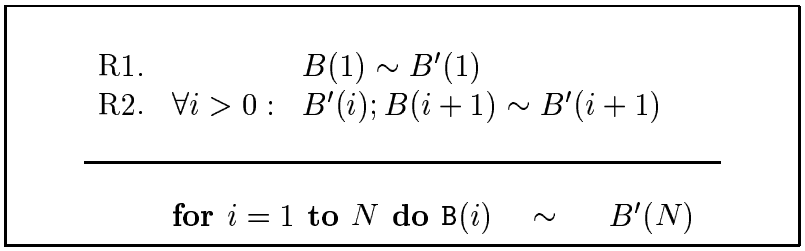

Figure 3.4: Rule REDUCE for loop reduction

Typically, 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. 3.3, $B(i)$ is $x := x + 1$ and $B'(i)$ is $x := x + i$. The two premises can easily be established for this simple case.

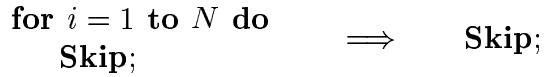

Figure 3.5: Reduction for an empty loop.

Rule REDUCE can also be used to show that a loop which does nothing can be removed. Fig. 3.5 shows a transformation which removes a loop with no loop body. In this case, $B(i) = B'(i) = \textbf{Skip}$.

3.3 Generalized VALIDATE Rule

Chapter 2 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. 3.6 gives an example of such a transformation performed by ORC.

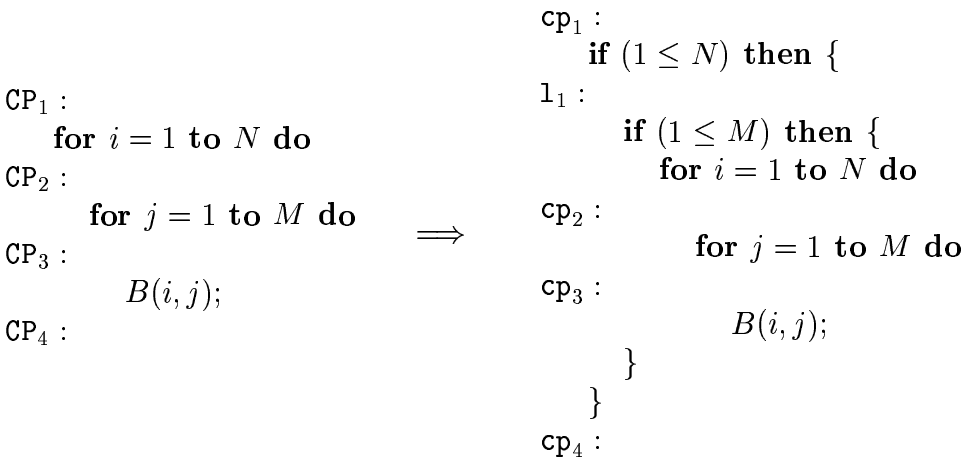

Figure 3.6: An example for which rule VALIDATE fails.

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.

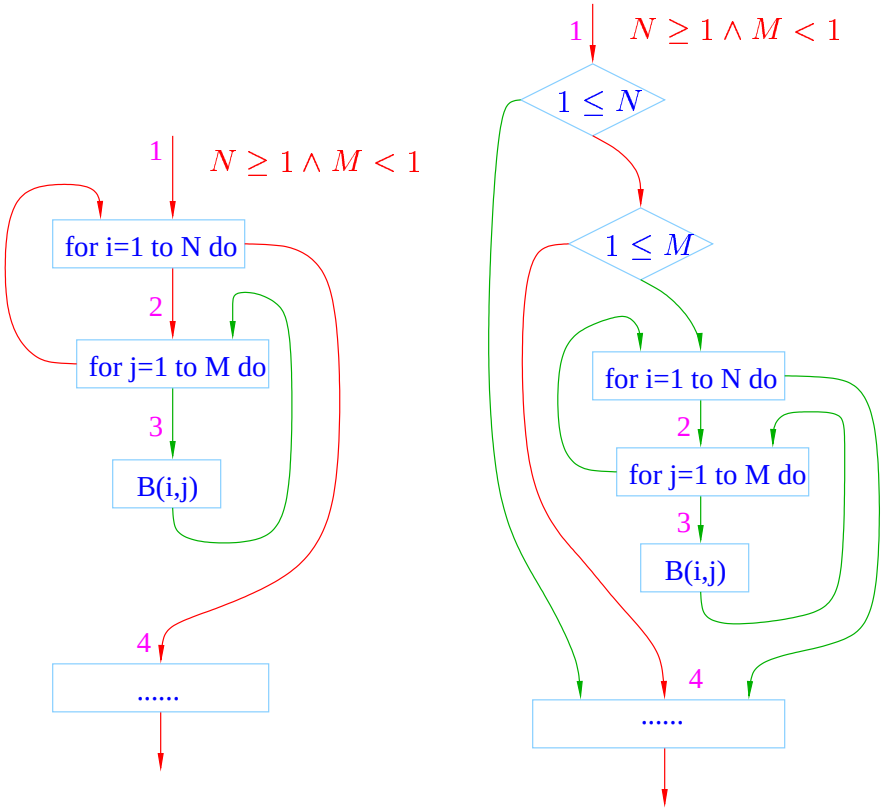

Figure 3.7: Rule GEN-VALIDATE example.

$\mathtt{l}_1$ is a target location that is not in the cut-point set. Now, consider the 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 (Fig. 3.7). 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 exactly the 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. 3.6 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 [HBGP05], is presented in Fig. 3.8, and a proof of its correctness is given in Section 3.5. It is essentially the same proof rule as that given in [ZPG+05] 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 be-

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* κ: $\mathsf{CP}_T \to \mathsf{CP}_S$ 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.

Figure 3.8: The generalized rule GEN-VALIDATE

fore, 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. 3.6, we can eliminate cut-points $\texttt{CP}_2$ and $\texttt{cp}_2$ as shown in Fig. 3.9. 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 $\texttt{CP}_1$ to $\texttt{CP}_4$. This path is only possible if the inner loop is never executed (otherwise $\texttt{CP}_3$ would be reached). But this means that the loop body is effectively empty, and as discussed earlier (see Fig. 3.5), 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 $\texttt{cp}_1$ to $\texttt{cp}_4$.

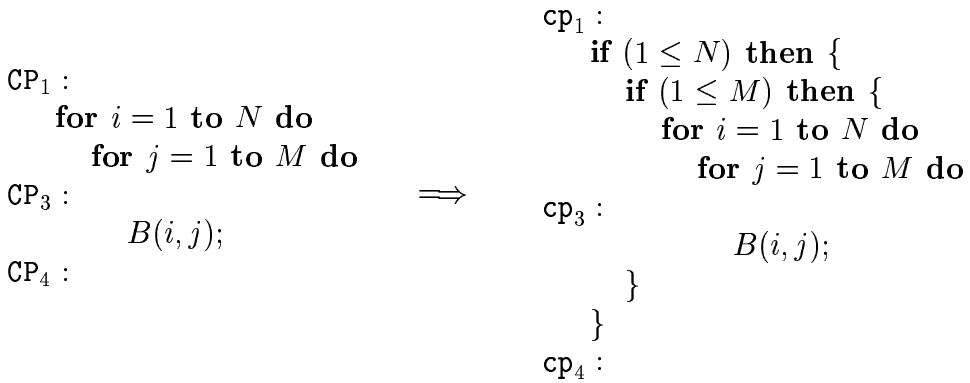

Figure 3.9: Example with modified cut-points.

3.4 Summary

This Chapter describes enhancements to our translation validation framework, primarily allowing additional loop and loop-related transformations to be validated. We introduced the new proof rule REDUCE, an improved permutation rule INV-PERMUTE, and a generalization of the old validate rule GEN-VALIDATE. Chapters 4 and 5 will describe the use of these rules in TVOC and speculative optimizations.

3.5 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, then $\sigma_{_T}$ 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,

$$C_{ij}: \qquad \varphi_i \ \wedge \ \alpha \ \wedge \ \rho^{_T}_{ij} \ \wedge \ \Big(\bigvee_{\pi \in Paths(\kappa(i))} \rho^{_S}_{\pi} \Big) \quad \rightarrow \quad \alpha' \ \wedge \ \varphi'_j.$$

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 $\bigvee_{\pi \in Paths(\kappa(i))} \rho^{_S}_{\pi}$ is always true. This condition guarantees that for the target simple path between i and j (it has computable transition relation $\rho^{_T}_{ij}$, and its corresponding source

simple path also has a computable transition relation ρ_π^S), 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.

Chapter 4

TVOC: A Translation Validator for Optimizating Compilers

TVOC is a tool for the translation validation of advanced optimizing compilers, using an automatic theorem prover CVC Lite [BB04]. In the framework of translation validation, to prove that the translated code correctly implements the original code, TVOC uses the proof rule VALIDATE for structure preserving optimizations with a clear mapping of control points between target and source programs (e.g. dead code reduction, loop-invariant code motion, copy propagation) [ARG99, WL91], and rule PERMUTE for reordering loop optimizations (e.g. interchange, tiling) which modify the iteration order of loops without modifying the loop bodies [AK02, Wolfe95].

This chapter is organized as follows. Section 1 introduces the architecture of TVOC, and provides an example of the tool. Section 2 gives the new implementation features in TVOC. Section 3 summarizes this chapter. In addition, Section 4 provides a sample output from TVOC.

4.1 Overview

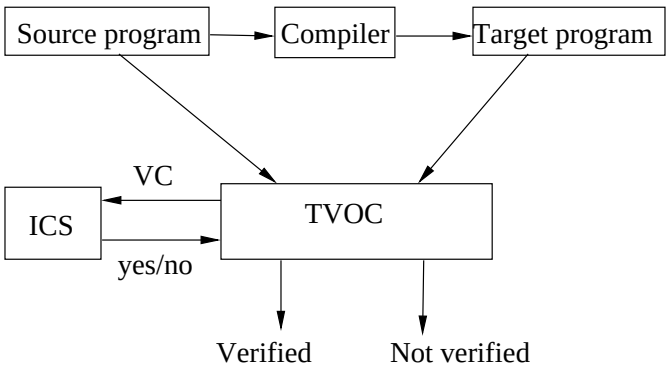

Figure 4.1: Initial design of TVOC.

The initial version of TVOC (Fig. 4.1) validated basic global optimizations employed by Intel's ORC compiler, using the ICS decision procedure [FORS01]. However, a lot of important issues were not considered initially: First, only global optimizations were dealt with, while other optimizations such as loop optimizations were not handled. Second, there were only scalar data types, while array types were not included. Third, as CVC Lite has advantages

of efficiency and the ability to produce verifiable proofs, it is a better choice for checking the verification conditions. To convert the decision procedure from ICS into CVC Lite, I used the commands and API interfaces of CVC Lite for types, expressions, assertions, queries, and contexts, and the details will not be presented here. The following paragraphs will describe how I improved TVOC considering the first two issues.

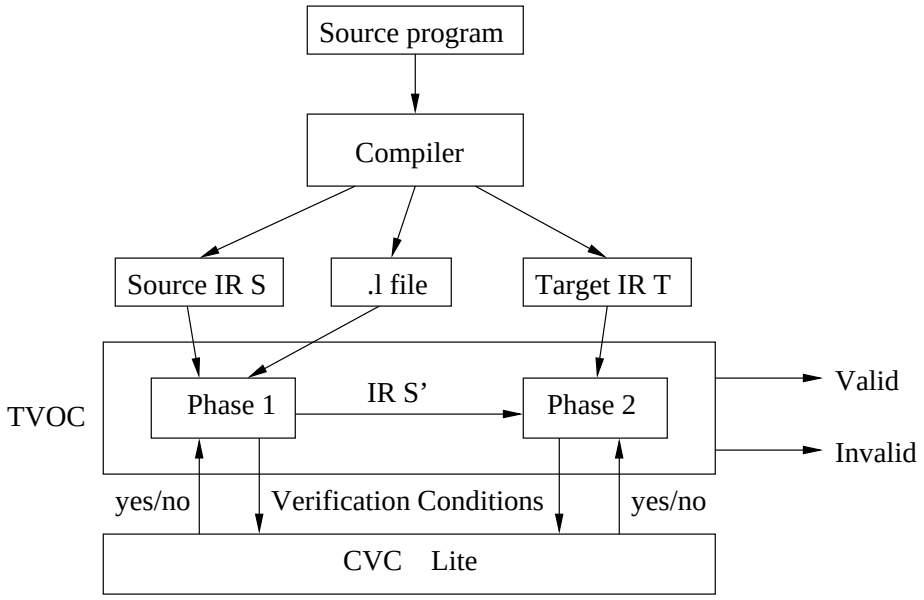

Figure 4.2: The second architecture of TVOC.

The second version of TVOC [HBGZ04] accepts as input the source intermediate code S and target intermediate code T generated from Intel's ORC compiler [JCW01], and outputs the result "VALID" with proof or "INVALID" with counter examples. This version was designed with two separate phases

(Fig. 4.2): the first phase implements rule PERMUTE for reordering loop optimizations and the second phase implements rule VALIDATE for structure preserving optimizations. These two phases are explained with the help of the following example, where two loops in the source are fused and an extra branch condition is added in the target.

```
do i = 0 to N                          if N ≥ 0 {
    A[i] = 0;                              do i = 0 to N {
                                               A[i] = 0;
                          ⟹                   B[i] = 1;
do i = 0 to N                              }
    B[i] = 1;                          }
```

In the first phase, in order to validate the loops, TVOC needs to know what kind of loop optimizations have actually been performed by the compiler. Since ORC compiler produces a .l file containing such information, this file was used in TVOC to get the hints from the compiler. Then TVOC generates verification conditions for these loop optimizations according to the proof rule PERMUTE, and validates them using CVC Lite. In the above example, for an iteration i_1 in the first loop and i_2 in the second loop, fusion reorders them when $i_2 < i_1$, so the rule requires:

$$i_2 < i_1 \quad \longrightarrow \quad A[i_1] = 0;\ B[i_2] = 1;\ \sim\ B[i_2] = 1;\ A[i_1] = 0;$$

Let $write(A, i, x)$ denote a new array obtained by "writing" x to the ith element of array A. The verification condition is represented as:

$$i_2 < i_1 \ \wedge \ A_1 = write(A, i_1, 0) \ \wedge \ B_1 = write(B, i_2, 1)$$
$$\wedge \ B'_1 = write(B, i_2, 1) \ \wedge \ A'_1 = write(A, i_1, 0)$$
$$\longrightarrow \qquad A_1 = A'_1 \ \wedge \ B_1 = B'_1$$

After validating the loops, TVOC produces a synthesized code S', which is the result from transforming the source code S with the guessed loop optimizations. For the above example, TVOC compares the source and the target, finds that two loops are fused, validates this fusion and produces a new S' code (Fig. 4.3).

```
do i = 0 to N
    A[i] = 0;

do i = 0 to N
    B[i] = 1;
```

(a) I_1 (Source S)

```
do i = 0 to N {
    A[i] = 0;
    B[i] = 1;
}
```

(b) I_2 (S')

```
if N ≥ 0 {
    do i = 0 to N {
        A[i] = 0;
        B[i] = 1;
    }
}
```

(c) I_3 (Target T)

Figure 4.3: S, S' and T for the example.

Source		Target	
		cp_0 :	IF $(n \geq 0)$
CP_0 :	DO (I:=0; $I \leq N$; I++)		DO (i:=0; $i \leq n$; i++)
CP_1 :	A[I] := 0;	cp_1 :	a[i] := 0;
	B[I] := 1;		b[i] := 1;
	ENDDO		ENDDO
CP_2 :			ENDIF
		cp_2 :	

Figure 4.4: Control points for the example.

In the second phase, after receiving the synthesized code S' from phase one, TVOC first finds the mapping α of variables between S' and the target code T, and transforms S' and T into flow graphs. TVOC then sets a series of control points CP_S and cp_T in the flow graphs, and builds the control mapping κ between them. TVOC also performs data flow analysis to get the invariants φ at the control points. Finally TVOC generates a set of verification conditions with α, κ and φ according to rule VALIDATE, and validates them using CVC Lite.

For the above example, both source S' and target T are marked at three points (Fig. 4.4). Assuming arrays A and B are observable variables, the proof task is to verify $A = a$ and $B = b$ at the terminal points (CP_2, cp_2) of the programs. The control mapping κ is:

$$cp_0 \mapsto CP_0 \quad cp_1 \mapsto CP_1 \quad cp_2 \mapsto CP_2.$$

Data mapping between source and target is:

$$PC = \kappa(pc) \wedge (A = a) \wedge (B = b).$$

There are four possible paths connecting two points directly $0 \rightarrow 1$, $0 \rightarrow 2$, $1 \rightarrow 1$ and $1 \rightarrow 2$. Therefore four verification conditions are established (Fig. 4.5), and they are verified by CVC Lite.

$$
\begin{array}{ll}
C_{01}: & True \ \wedge \ i' = 0 \ \wedge \ a' = a \ \wedge \ b' = b \ \wedge \ n' = n \ \wedge \ n \geq 0 \\
& \wedge \ A' = A \ \wedge \ B' = B \ \wedge \ I' = 0 \ \wedge \ N' = N \\
& \wedge \ a = A \ \wedge \ b = B \ \wedge \ i = I \ \wedge \ n = N \\
& \longrightarrow \quad a' = A' \ \wedge \ b' = B' \ \wedge \ i' = I' \ \wedge \ n' = N' \ \wedge \ i' \leq n' \ \wedge \ I' \leq N' \\
C_{02}: & True \ \wedge \ i' = i \ \wedge \ a' = a \ \wedge \ b' = b \ \wedge \ n' = n \ \wedge \ n < 0 \\
& \wedge \ A' = A \ \wedge \ B' = B \ \wedge \ I' = 0 \ \wedge \ N' = N \\
& \wedge \ a = A \ \wedge \ b = B \ \wedge \ i = I \ \wedge \ n = N \\
& \longrightarrow \quad a' = A' \ \wedge \ b' = B' \ \wedge \ n' = N' \ \wedge \ True \\
C_{11}: & i \leq n \ \wedge \ I \leq N \\
& \wedge \ i' = i + 1 \ \wedge \ a' = write(a, i, 0) \ \wedge \ b' = write(b, i, 1) \ \wedge \ n' = n \\
& \wedge \ i + 1 \leq n \ \wedge \ A' = write(A, I, 1) \ \wedge \ B' = write(B, I, 1) \\
& \wedge \ I' = I + 1 \ \wedge \ N' = N \wedge \ a = A \ \wedge \ b = B \ \wedge \ i = I \ \wedge \ n = N \\
& \longrightarrow \quad a' = A' \ \wedge \ b' = B' \ \wedge \ i' = I' \ \wedge \ n' = N' \ \wedge \ i' \leq n' \ \wedge \ I' \leq N' \\
C_{12}: & i \leq n \ \wedge \ I \leq N \\
& \wedge \ i' = i + 1 \ \wedge \ a' = write(a, i, 0) \ \wedge \ b' = write(b, i, 1) \ \wedge \ n' = n \\
& \wedge \ i + 1 > n \ \wedge \ A' = write(A, I, 1) \ \wedge \ B' = write(B, I, 1) \\
& \wedge \ I' = I + 1 \ \wedge \ N' = N \ \wedge \ a = A \ \wedge \ b = B \ \wedge \ i = I \ \wedge \ n = N \\
& \longrightarrow \quad a' = A' \ \wedge \ b' = B' \ \wedge \ n' = N' \ \wedge \ I + 1 > N
\end{array}
$$

Figure 4.5: Verification conditions for the example.

For a given source program, if both phases of TVOC yield positive results, it is guaranteed by the proof rules that the source and target intermediate codes are equivalent. For the above example, TVOC outputs result "Valid",

which means that the source and target are indeed equivalent.

4.2 New Implementation Features

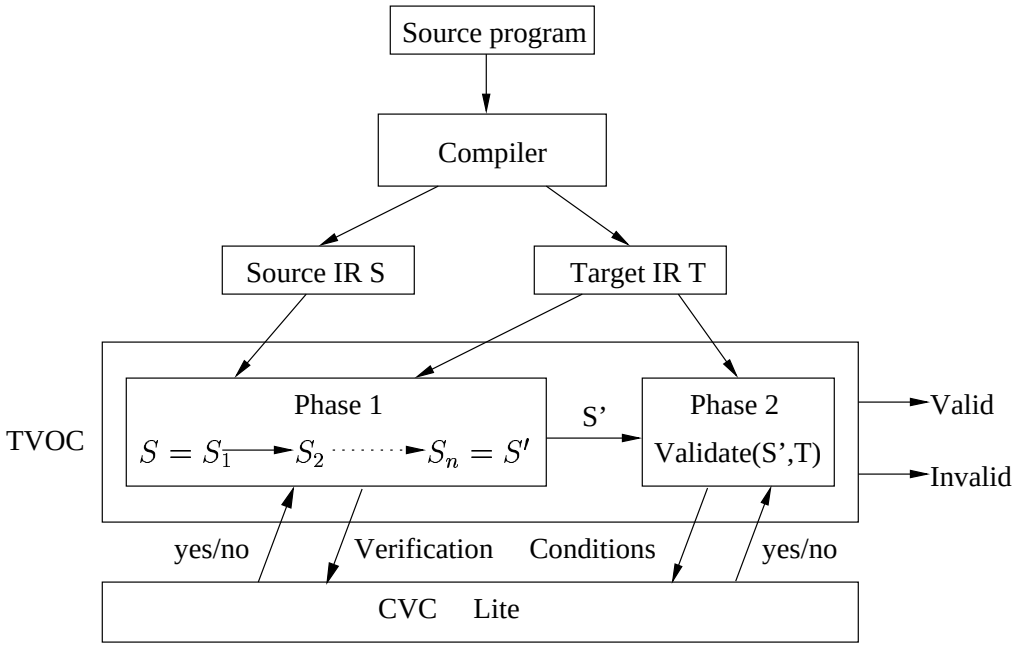

Figure 4.6: The current architecture of TVOC.

Fig. 4.2 shows the previous architecture of TVOC. There are three limitations of this architecture [GZB04]: first, TVOC depended on an auxiliary file (the ".l" file) generated by the compiler to tell it which loop transformations had been performed; second, loop transformations had to be verified in one step, even if the transformations were more naturally modeled as the compo-

sition 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 [BFG+05] (shown in Fig. 4.6) and show how we implemented the solutions to these problems. We also discuss the implementation of the GEN-VALIDATE rule described in Chapter 3.

4.2.1 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 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. 4.7.

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.

Figure 4.7: The algorithm for analyzing loop transformations.

4.2.2 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 (Fig. 4.8). This was not ideal from a software engineering perspective. In the current version, TVOC uses the generalized approach described in [GZB04] for all reordering loop transformations (Fig. 4.9). 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.

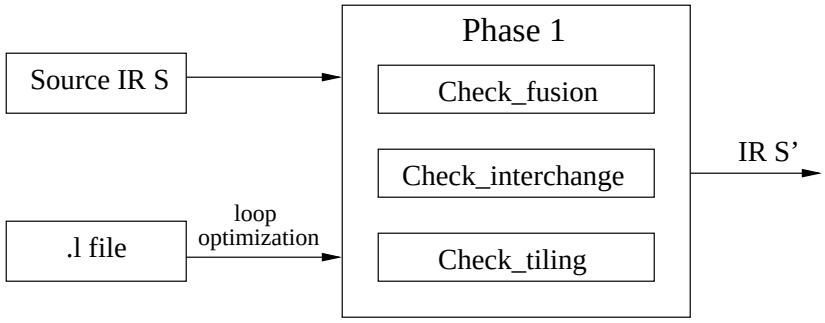

Figure 4.8: Previous design with different validation modules.

4.2.3 A Methodology for Combinations of Optimizations

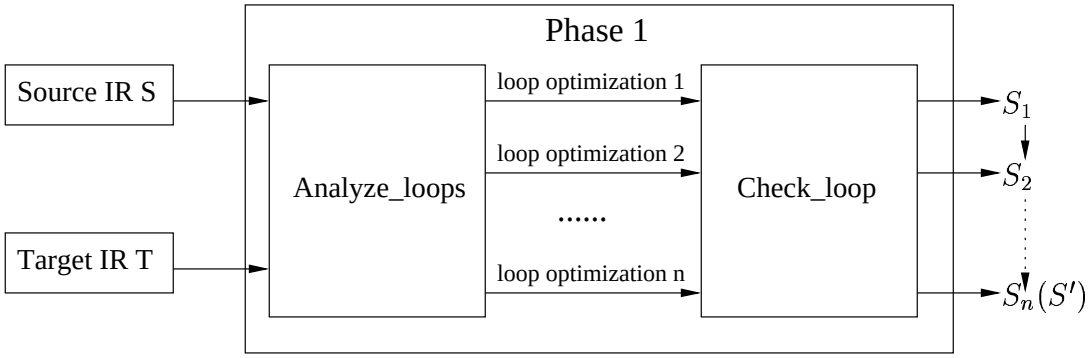

Figure 4.9: Combinations of optimizations.

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 series of intermediate versions of codes $S_1, S_2, ...S_n$ are 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. 4.9).

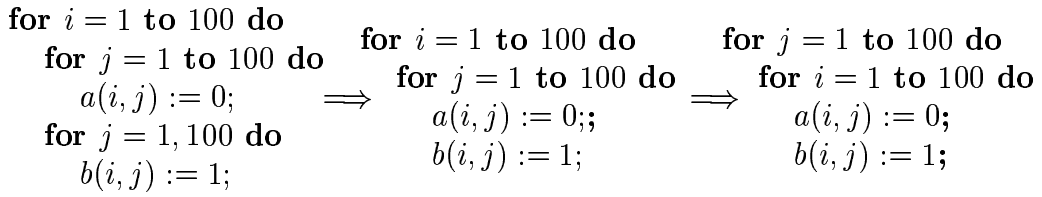

Figure 4.10: A combination of loop transformations.

As an example, consider the code in Fig. 4.10. 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.

4.2.4 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. The verification condition as shown 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 $Cond_\pi^S$ be the conditions under which a source simple path π is enabled (this corresponds to a conjunction of the branch conditions along the path).

Theorem 4.2.1 *Consider the following verification conditions:*

$$\varphi_i \ \wedge \ \alpha \ \wedge \ \rho_{ij}^T \ \wedge \ \Big(\bigvee_{\pi \in Paths(\kappa(i))} \rho_\pi^S \Big) \quad \rightarrow \quad \alpha' \ \wedge \ \varphi'_j, \tag{4.1}$$

$$\varphi_i \ \wedge \ \alpha \ \wedge \ \rho_{ij}^T \quad \rightarrow \quad \Big(\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} Cond_\pi^S \Big) \tag{4.2}$$

$$\varphi_i \ \wedge \ \alpha \ \wedge \ \rho_{ij}^{T} \ \wedge \ (\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} \rho_{\pi}^{S}) \quad \rightarrow \quad \alpha' \ \wedge \ \varphi_j', \tag{4.3}$$

We claim that equation (4.1) holds iff (4.2) and (4.3) hold.

Proof In one direction, suppose (4.1) holds, (4.3) also holds because the left-hand side of (4.3) is stronger than the left-hand side of (4.1) while the right-hand sides of the two implications are the same. Now, to show that (4.2) also holds, suppose we have $\varphi_i \ \wedge \ \alpha \ \wedge \ \rho_{ij}^{T}$. 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 $(\bigvee_{\pi \in Paths(\kappa(i))} \rho_{\pi}^{S})$ holds. By (4.1), we then have $\alpha' \ \wedge \ \varphi_j'$. But from $\rho_{ij}^{T} \wedge \alpha'$, $\mathtt{PC}' = \kappa(j)$ follows. We thus have $(\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} \rho_{\pi}^{S})$, which means (4.2) holds since $Cond_{\pi}^{S}$ is implied by ρ_{π}^{S}.

In the other direction, suppose that (4.2) and (4.3) hold and that we have $\varphi_i \wedge \alpha \wedge \rho_{ij}^{T} \wedge (\bigvee_{\pi \in Paths(\kappa(i))} \rho_{\pi}^{S})$. By (4.2), we have $(\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} Cond_{\pi}^{S})$, 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 $(\bigvee_{\pi \in Paths(\kappa(i),\kappa(j))} \rho_{\pi}^{S})$ holds. By (4.3), we have $\alpha' \ \wedge \ \varphi_j'$, and thus (4.1) 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 $\kappa(i)$.

4.2.5 Introducing array types

Loops often contain large arrays, and they are hard to analyze due to memory aliasing. In TVOC, arrays of different names are regarded as different arrays with no overlap in memory locations, assuming the program is free of pointers. In this case, memory aliasing is restricted within an array.

In data flow analysis, a basic block is a sequence of statements with an unique entry and exit, thus there is no branch inside a basic block. All statements in a basic block B_i are considered together to get the combined transition relation ρ_i for this basic block. With only scalar data types, this combined transition relation ρ_i is represented by a variable map which establishes a function m_i mapping the name of a variable v to the value of the variable $m_i(v)$. Each mapping has $m_i(v) = v$ initially, and it is updated into m_i' by the assignments in the block. The combined transition relation ρ_i is represented with

$$\bigwedge_{v \in V} v' = m_i'(v)$$

where V is the set of all variables in the program, and v' holds the value of v after the block. The composed result of two basic blocks B_i and B_j can be

computed by composing their mapping m_i and m_j:

$$\bigwedge_{v \in V} v' = m'_j(m'_i(v)).$$

To show how to compute the mapping, consider an arbitrary basic block with a variable mapping m, it is assumed that the mapping m satisfies:

$$m(e_1 \ op \ e_2) = m(e_1) \ op \ m(e_2)$$

$$m(op \ e) = op \ m(e)$$

$$m(C) = C$$

where operator *op* stands for any scalar operation, and C is a constant. For a single assignment $x = e_x;$, the mapping function m is updated by m^1 with

$$m^1(v) = m(v) \qquad if \quad v \neq x;$$

$$m^1(x) = m(e_x).$$

If there is another assignment $y = e_y;$ following the above assignment, then

after another update, the mapping becomes m^2:

$$m^2(v) = m^1(v) \qquad if \quad v \neq y;$$

$$m^2(y) = m^1(e_y).$$

In this way, the final mapping m' of the basic block is computed, by updating the initial mapping m with the statements sequentially.

The above algorithm considers scalars only. Arrays are difficult to handle because each array assignment only modifies an element of an array, while preserving all the other elements. Some extra expressions are needed to model the special properties of array updates. Following CVC Lite, two operators *read* and *write* are introduced:

$read(a, i)$ means the value of element i of array a.

$write(a, i, e)$ means the element i of array a is updated with value e.

It is assumed that any mapping m satisfies:

$$m(read(a, i)) = read(m(a), m(i))$$

$$m(write(a, i, e)) = write(a, m(i), m(e)).$$

And initially the mapping m has:

$$m(a) = a$$

for any array a in the program.

With the above representations, the array-read assignment $x = a[i];$ updates the map m with

$$m'(x) = read(m(a), m(i));$$

And the array-write assignment $a[i] = e;$ updates the map m with

$$m'(a) = write(a, m(i), m(e)).$$

With these definitions, the behavior of arrays is modeled correctly. For example, if the first statement in the block is

$$a[i + 1] = a[i] + 1;$$

then the map m is updated into m^1, where

$$\begin{aligned} m^1(a) &= write(a, m(i+1), m(read(a,i)+1)) \\ &= write(a, m(i)+m(1), m(read(a,i))+m(1)) \\ &= write(a, i+1, read(m(a), m(i))+1) \\ &= write(a, i+1, read(a,i)+1). \end{aligned}$$

4.3 Summary

This chapter gives the implementation details on the translation validation tool based on the theory presented in Chapter 2 and 3. The two-phase architecture of TVOC was introduced first, then the implementation features were described focused on the part for loop optimizations.

TVOC can handle a significant variety of compiler optimizations including combinations of both reordering transformations and structure-preserving optimizations. At this point, TVOC still has some limitations. There are still some optimizations cannot be validated yet. However, our experience is that TVOC is successful most of the time. The current version of TVOC does not handle procedures, aliasing, or pointers. However, we expect to handle these features in the future.

Although TVOC has primarily been used as a research prototype and ex-

perimental platform for theoretical work, we are hoping it will be of use and interest to a broader community. It is available together with basic examples and documentation at
http://www.cs.nyu.edu/acsys/tv/.

4.4 Appendix: Sample output from TVOC

```
First pass:
-----------
-----------
Source input code:
----
MAIN__()
 {
 For (   I := 0;   I <= t264 ;   I := I + 1; ) Do
  {
  (A[I + (-1)]) := 0;
  }
 For (   I := 0;   I <= t265 ;   I := I + 1; ) Do
  {
  (B[I + (-1)]) := 1;
```

```
  }
 }
%% Fusion:
% The loop before optimization:
 For (   I := 0;   I <= t264 ;   I := I + 1; ) Do
  {
  (A[I + (-1)]) := 0;
  }
% The second loop of fusion:
 For (   I := 0;   I <= t265 ;   I := I + 1; ) Do
  {
  (B[I + (-1)]) := 1;
  }
% Before loop optimization:
(A#1 = (A WITH [(I_1 + -(1))] := 0)).
(B#1 = (B WITH [(I_2 + -(1))] := 1)).
% After loop optimization:
(B#1p = (B WITH [(I_2 + -(1))] := 1)).
(A#1p = (A WITH [(I_1 + -(1))] := 0)).
% (I_1) >LEX (I_2):
(I_1 > I_2).
```

```
Check A#1 = A#1p

%% Valid

Check B#1 = B#1p

%% Valid


% The loop after optimization:

 For (   I := 0;   I <= t264 ;   I := I + 1; ) Do

  {

  (A[I + (-1)]) := 0;

  (B[I + (-1)]) := 1;

  }

% The function after fusion:

MAIN__()

 {

 For (   I := 0;   I <= t264 ;   I := I + 1; ) Do

  {

  (A[I + (-1)]) := 0;

  (B[I + (-1)]) := 1;

  }

 }


Second pass:
```

```
-----------

-----------

Source code:

----

B0 (S0) --

 I := 0;

B1 --

 For (   I := 0;   I <= M ;   I := I + 1; ) Do

 {

B2 (S2) --

  (A[I + (-1)]) := 0;

  (B[I + (-1)]) := 1;

 }

B3 (S3) --


Target code:

----

B0 (T0) --

 If (M >= 0)

 {

B1 --

  .t268 := 0;
```

```
B2 --
  For (    .t268 := 0;   .t268 <= M ;    .t268 := .t268 + 1; ) Do
  {
B3 (T3) --
   (A[.t268 + (-1)]) := 0;
   (B[.t268 + (-1)]) := 1;
  }
 }
 Else{
 }
 EndIf
B4 (T4) --
----
Ccontrol Mapping:
         From T0 To S0
         From T3 To S2
         From T4 To S3
Variable List:
 Source:
A       B       I       M
 Target:
.t268   A       B       M
```

```
### CVC-Lite starts:

Result:

C(0, 3)  is VALID.

C(0, 4)  is VALID.

C(0, 4)  is VALID.

C(3, 3)  is VALID.

C(3, 4)  is VALID.

### CVC-Lite succeeds checking.


Data Mapping:

alpha =

   at point T0 :

        (A_t = A_s)

        (B_t = B_s)

        (.t268_t = I_s)

        (M_t = M_s)

   at point T3 :

        (A_t = A_s)

        (B_t = B_s)

        (.t268_t = I_s)

        (M_t = M_s)
```

```
   at point T4 :

        (A_t = A_s)

        (B_t = B_s)

        (M_t = M_s)


C(0, 3)  VC:

   inv =

   target transitions =

        (.t268_t' = 0)

        (A_t' = A_t)

        (B_t' = B_t)

        (M_t' = M_t)

   source transitions =

        (((((0 <= M_s) AND (A_s' = A_s)) AND (B_s' = B_s))

        AND (I_s' = 0)) AND (M_s' = M_s))

   target conditions =

        (M_t >= 0)

        (0 <= M_t)

   alpha =

        (A_t = A_s)

        (B_t = B_s)

        (.t268_t = I_s)
```

```
        (M_t = M_s)
    ===>
    alpha' =
        (A_t' = A_s')
        (B_t' = B_s')
        (.t268_t' = I_s')
        (M_t' = M_s')
    inv' =
        (.t268_t' <= M_t')
        (I_s' <= M_s')

C(0, 4)  VC:
    inv =
    target transitions =
        (.t268_t' = .t268_t)
        (A_t' = A_t)
        (B_t' = B_t)
        (M_t' = M_t)
    source transitions =
        (((((0 > M_s) AND (A_s' = A_s)) AND (B_s' = B_s))
        AND (I_s' = 0)) AND (M_s' = M_s))
    target conditions =
```

```
        (M_t < 0)
    alpha =
        (A_t = A_s)
        (B_t = B_s)
        (.t268_t = I_s)
        (M_t = M_s)
    ===>
    alpha' =
        (A_t' = A_s')
        (B_t' = B_s')
        (M_t' = M_s')
    inv' =

C(0, 4)  VC: Not viable!
    inv =
    target transitions =
        (.t268_t' = 0)
        (A_t' = A_t)
        (B_t' = B_t)
        (M_t' = M_t)
    source transitions =
    target conditions =
```

```
        (M_t >= 0)

        (0 > M_t)

    alpha =

        (A_t = A_s)

        (B_t = B_s)

        (.t268_t = I_s)

        (M_t = M_s)

    ===>

    alpha' =

        (A_t' = A_s')

        (B_t' = B_s')

        (M_t' = M_s')

    inv' =


C(3, 3)  VC:

    inv =

        (.t268_t <= M_t)

        (I_s <= M_s)

    target transitions =

        (.t268_t' = (.t268_t + 1))

        (A_t' = (A_t WITH [(.t268_t + -1)] := 0))

        (B_t' = (B_t WITH [(.t268_t + -1)] := 1))
```

```
     (M_t' = M_t)
source transitions =
     ((((((I_s + 1) <= M_s)
     AND (A_s' = (A_s WITH [(I_s + -1)] := 0)))
     AND (B_s' = (B_s WITH [(I_s + -1)] := 1)))
     AND (I_s' = (I_s + 1))) AND (M_s' = M_s))
target conditions =
     ((.t268_t + 1) <= M_t)
alpha =
     (A_t = A_s)
     (B_t = B_s)
     (.t268_t = I_s)
     (M_t = M_s)
===>
alpha' =
     (A_t' = A_s')
     (B_t' = B_s')
     (.t268_t' = I_s')
     (M_t' = M_s')
inv' =
     (.t268_t' <= M_t')
     (I_s' <= M_s')
```

```
C(3, 4)  VC:
   inv =
        (.t268_t <= M_t)
        (I_s <= M_s)
   target transitions =
        (.t268_t' = (.t268_t + 1))
        (A_t' = (A_t WITH [(.t268_t + -1)] := 0))
        (B_t' = (B_t WITH [(.t268_t + -1)] := 1))
        (M_t' = M_t)
   source transitions =
        ((((((I_s + 1) > M_s)
        AND (A_s' = (A_s WITH [(I_s + -1)] := 0)))
        AND (B_s' = (B_s WITH [(I_s + -1)] := 1)))
        AND (I_s' = (I_s + 1))) AND (M_s' = M_s))
   target conditions =
        ((.t268_t + 1) > M_t)
   alpha =
        (A_t = A_s)
        (B_t = B_s)
        (.t268_t = I_s)
        (M_t = M_s)
```

```
===>

alpha' =

     (A_t' = A_s')

     (B_t' = B_s')

     (M_t' = M_s')

inv' =
```

Chapter 5

Speculative Loop Optimizations

A modern compiler performs a set of advanced optimizations to make the compiled code run faster. Among them are loop optimizations which improve parallelism and make efficient use of the memory hierarchy. 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 [BB04], 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 chapter gives an algorithm for generating such a condition using CVC Lite.

In some cases, it is impossible to determine at compilation time whether a

desired loop optimization is legal. This is usually because of limited capability to check effectively that syntactically different array index expressions refer to the same array location. In such cases, the validation condition derived by CVC Lite cannot be proved to hold at compile-time, but it may hold at run time. One possible remedy to this situation is to perform the optimization conditionally, by adding code to check at run time whether the loop optimization is safe. If the run-time check fails, the code chooses to use an unoptimized version of the loop which completes the computation in a manner which may be slower but is guaranteed to be correct. An algorithm for generating the run-time test for speculative loop optimizations was given in [BGZ03]. The proof rule INV-PERMUTE was introduced in Chapter 3 as the formal basis for using loop invariants to validate loop optimizations (both speculative and non-speculative). The algorithm for finding invariants needed to apply rule INV-PERMUTE has been improved and will be described in more detail in this chapter. More examples and results will be given.

This chapter is organized as follows. Section 5.1 introduces speculative loop optimizations. Section 5.2 describes an improved algorithm for determining a sufficient condition under which an otherwise invalid transformation may be applied. Using the proof rule, we show that such a transformation is valid if the condition can be statically verified. Alternatively, a run-time test for the condition can be inserted. Section 5.3 gives several examples and shows the

results of applying the algorithms in Section 5.2 to these examples. Finally, Section 5.4 concludes.

5.1 Introduction

Sometimes the compiler has to behave conservatively when it cannot decide the validity of some loop optimizations because the values of some variables are not known at compile time. For these cases, speculative loop optimizations [BGZ03] can introduce runtime tests on the unknown variables into the compiled code so that the loop optimizations will be performed if the tests are satisfied at run-time.

It was proposed [BGZ03] to use the automatic theorem prover CVC Lite to help the compiler decide the validity of loop transformations and construct the run-time tests automatically. That means, the compiler can generate the verification conditions (VCs) for the optimizations to be valid, input the VCs to CVC Lite, and get results back. If CVC Lite reports valid, then the loop transformations will be performed under all contexts; Otherwise the conditions reported by CVC Lite will be used as run-time tests, and the loop transformations will be performed conditionally.

A strategy was proposed [BGZ03] to use the counter examples reported by CVC Lite to derive conditions that ensure the correctness of the optimizations.

Assuming ϕ is the initially invalid verification condition, the algorithm is listed as follows:

0. Let $\psi = \emptyset$.
1. Check $\bigwedge(\psi) \rightarrow \phi$ using CVC Lite.
2. If the result is valid, exit.
3. Use the `WHERE` command to obtain a set of assumptions θ under which the formula ϕ is false.
4. Select a formula from θ, negate it, and add it to ψ.
5. Goto 1.

The loop is exited with a set of conditions ψ under which the optimization is valid.

The difficult part about automatically producing the conditions is how to efficiently choose a formula from the counter examples reported by CVC Lite. Heuristics for it will be presented in Section 5.2.

5.2 The algorithm for using the proof rule INV-PERMUTE

This section gives the algorithms for using the proof rule INV-PERMUTE to validate loop transformations and to generate run-time tests for speculative

loop optimizations.

5.2.1 Loop transformations with invariants

The compiler can decide whether some loop transformation is valid on the basis of the INV-PERMUTE rule and the static analysis of the initial condition and the invariant condition of the loop. For a given loop transformation, the function F is known, but the precondition ϕ can be any condition that is implied by the initial condition of the loop. An initial condition ϕ_0 can be determined from dataflow analysis, but it may be too strong. The INV-PERMUTE rule only needs an invariant condition ϕ that makes premise 4 valid, which suggests finding an appropriate ϕ by trying to validate premise 4. If there is no ϕ satisfying premise 4, then the requirements of INV-PERMUTE cannot be satisfied. Thus, a feasible method is to first analyze premise 4 to find a condition ϕ under which it is valid, then check this ϕ to see whether it is implied by ϕ_0 and preserved by the loop body. The main probem becomes how to find this ϕ which makes premise 4 valid. Since the theorem prover CVC Lite is able to check the validity of formulas and generate counter-examples efficiently, it can be used for the purpose of finding ϕ. With the help of CVC Lite, the scheme for validating reordering loop optimizations is:

1. Apply INV-PERMUTE for the loop under $\phi = true$, and generate the verification condition θ for premise 4.

2. Check the validity of θ using CVC Lite. If it is valid, exit with a positive result.

3. Otherwise, from the counter-example ψ produced by CVC Lite, attempt to infer a condition ϕ that makes the verification condition valid. If no appropriate ϕ is found exit with a negative result.

4. Analyze the context statically to check whether ϕ holds as the initial condition and is loop invariant. If ϕ holds, exit with a positive result.

5. Otherwise, exit with a negative result.

Step 3 will be explained in more detail in Section 5.2.3.

In this scheme, $\phi = true$ is used initially to avoid the analysis for the initial loop condition when possible. The verification condition (VC) according to premise 4 is input to CVC Lite. If CVC Lite reports valid, then the loop transformation is valid under all contexts. Otherwise the counter-example reported by CVC Lite can be analyzed to construct a candidate condition ϕ. If the new ϕ holds as a precondition and is invariant in the loop, and if premise 4 is satisifed, then the reordering loop transformation is valid.

5.2.2 Speculative loop optimizations

The INV-PERMUTE rule requires that ϕ hold on entry to the loop (both original and transformed versions). However, if the values of some variables are not known at compile time, information about them cannot be included in ϕ. In such cases, a loop optimization might not be able to be validated using only compile-time information, but the optimization might actually be valid at run-time.

```
if (k ≥ 0)
    for j = 1 to M
        for i = 1 to N
                A[i+k,j+1] = A[i,j] + 1;
else
    for i = 1 to N
        for j = 1 to M
                A[i+k,j+1] = A[i,j] + 1;
```

Figure 5.1: An example for speculative loop interchange

To preserve the benefit of loop optimizations in the presence of variables whose values cannot be determined statically, a run-time test, testing the values of various variables used in the loop, can be inserted into the compiled program. Loop optimizations enabled in this manner are called *speculative* loop optimizations[1]. The idea of validating speculative loop optimizations in the TV framework was introduced in [BGZ03]. However since the INV-

[1]This technique was called *inspector/executor* in [BS90, PR98].

PERMUTE rule was not established in that paper, only the concept and some heuristics were given. In this thesis, Chapter 3 described the new proof rule INV-PERMUTE, which is the formal basis for validating speculative loop optimizations. Fig. 5.1 shows the result of applying a speculative loop optimization to the interchange example of Fig. 3.1.

With the INV-PERMUTE rule, the scheme for speculative loop optimizations is:

1. Apply INV-PERMUTE for the loop, using $\phi = true$, and generate the verification condition θ for premise 4.
2. Check the validity of θ using CVC Lite. If it is valid, exit with a positive result.
3. Otherwise, from the counter-example ψ produced by CVC Lite, attempt to infer a condition ϕ that makes the verification condition valid. If no appropriate ϕ is found exit with a negative result.
4. Analyze the context statically to check whether ϕ holds as the initial condition and is loop invariant. If ϕ holds, exit with a positive result.
5. Otherwise, if ϕ is satisfiable (i.e. it could hold under some run-time conditions), is inductive in the loop, and is not too

costly to evaluate, exit and use ϕ to generate a run-time test for a speculative loop optimization; else exit with a negative result.

5.2.3 Automatically generating invariants using CVC Lite

We implement Step 3 in the previous algorithms as follows:

0. Let $\phi = true$.
1. Check $\phi \rightarrow \theta$ using CVC Lite.
2. If the result is valid, exit with ϕ.
3. From the counter-example $\psi = \bigwedge_i (C_i)$, choose an appropriate subset S for i, and let $\phi = \phi \wedge \neg(\bigwedge_{i \in S} C_i)$.
4. Goto 1.

Since we choose counter-example assertions until we have a sufficient condition ϕ, the invariant ϕ we get may be stronger than necessary. To avoid ϕ being either too strong or too complicated, good heuristics need to be used to pick the appropriate C_is from a set of formulas.

The following are some observations: As the invariant ϕ must refer to non-index variables, at least one such variable must be in the chosen formula. Because equality is usually a stricter requirement than disequality, the chosen

formula should not be a disequality such as $\neg(x = y)$. Also a formula including array elements may not be a good choice (as they generally include index variables that may be hard to eliminate).

For the validity of interchanging the loop example in Fig. 3.1, CVC Lite generates the following counter example with six formulas:

1.) $0 < i_2 - i_1$

2.) $(i_2 + k, 1 + j_2) = (i_1, j_1)$

3.) NOT $(i_2 + k, 1 + j_1) = (i_2, j_2)$

4.) NOT $(-k = 0)$

5.) NOT $(i_2 + k, 1 + j_1) = (i_2 + k, 1 + j_2)$

6.) NOT $(i_2, j_1) = (i_2 + k, 1 + j_2)$

Since formula 1 does not include any non-index variables, and formulas 3, 4, 5, 6 are disequality formulas, none of them are candidates according to our heuristics. The only choice is formula 2. We get the result $\phi = \neg((i_2 + k, j_2 + 1) = (i_1, j_1))$ using our algorithm. This ϕ is exactly the same as what would result from dependence analysis. Using this value for ϕ, premise 4 of rule INV-PERMUTE is valid. The problem is that this ϕ is still not useful, since the invariant in INV-PERMUTE is assumed to be independent of loop index variables. So we have to find some way to eliminate the loop index variables

from ϕ. To do this, we use the constraints on the loop index variables (i.e. the loop bounds and the order of iterations) to aid in removing of loop index variables, as explained below.

Let θ denote the verification condition derived from premise 4 of rule INV-PERMUTE with invariant ϕ. θ can be divided into three parts: the first part is a constraint on the loop index variables which includes the loop bound and the condition of reordering, let's denote it as α; the second part is the invariant ϕ; and the third part is the formula for equivalence of executing the two iterations in both orders, let's denote it as β. So the formula θ can be expressed as $\alpha \rightarrow \phi \rightarrow \beta$. This formula is equivalent to:

$$(\phi \wedge \alpha) \rightarrow \alpha \rightarrow \beta.$$

Assume we find a condition ϕ' stronger than ϕ under α, i.e. $\alpha \rightarrow \phi' \rightarrow \phi$, then if θ is valid, it is guaranteed that

$$(\phi' \wedge \alpha) \rightarrow \alpha \rightarrow \beta$$

is also valid, which is equivalent to $\phi' \rightarrow \alpha \rightarrow \beta$. Thus, as long as we find a formula ϕ' which is stronger than ϕ under α, using this ϕ' instead of ϕ can also ensure the validity of the loop transformations.

The condition ϕ generated from CVC Lite is always some disequality or inequality since the heuristic we use discards equality. Since, in most cases, the expressions for the subscripts of array elements are linear, ϕ can be assumed to be in the form $e_1\ OP\ e_2(\vec{i_1}, \vec{i_2})$ where e_1 is an expression free of loop index variables, and e_2 is an expression containing no variables except loop index variables, and the relational operator $OP \in \{>, <, \geq, \leq, \neq\}$. As long as e_2 is linear in the loop variables, α can be used to eliminate the loop variables from e_2. Based on the above observations, an algorithm was designed to derive a ϕ' free of loop index variables from ϕ and α. For the above interchange example, from $\phi = \neg((i_2 + k, j_2 + 1) = (i_1, j_1))$, after using the algorithm eliminating the loop index variables with

$\alpha:\ i_1 - i_2 < 0\ \wedge\ i_1 - i_2 > -N$

$\wedge\ j_1 - j_2 > 0\ \wedge\ j_1 - j_2 < N$

and

$\beta:\ A_1 = write(A, (i_1 + k, j_1 + 1), A[i_1, j_1] + 1)$

$\wedge A_2 = write(A_1, (i_2 + k, j_2 + 1), A_1[i_2, j_2] + 1)$

$\wedge A'_1 = write(A, (i_2 + k, j_2 + 1), A[i_2, j_2] + 1)$

$\wedge A'_2 = write(A'_1, (i_1 + k, j_1 + 1), A'_1[i_1, j_1] + 1)$

$\longrightarrow \qquad A_2 = A'_2,$

we arrive at ϕ' being $k \geq 0\ \vee\ k \leq -N$. This condition is checked again by CVC Lite to make sure the verification condition is valid.

5.3 Results

We have implemented our algorithm to generate the invariant ϕ' using CVC Lite for loop optimizations such as fusion, interchange, reversal and tiling. The conditions ϕ and ϕ' were generated automatically for all the (small) examples we tested. This section gives the results for the following four examples.

```
for i = 1 to N                          for i = 1 to N
        A[i] = i;          ⟹                   A[i] = i;
for i = 1 to N                                  y = A[i-k];
        y = A[i-k];
```

Figure 5.2: A fusion example

The first example is the fusion example in Fig. 5.2, where two simple loops in the source are merged into one simple loop in the target. Observe that this transformation is legal when no anti-dependence is created during the merging of the loops, i.e it is legal when $(k \geq 0)$ or $(|k| \geq N)$. Notice that the two loops in the source can be treated as a general loop structure [GZB04], such that rule INV-PERMUTE can be applied. The following table gives the logical formulas for α, β, ϕ, and ϕ'. The result shows that the fusion is valid when $k \geq 0 \ \vee \ k \leq -N$, which is consistent with the above observation.

$$\alpha: \quad i_1 - i_2 > 0 \ \wedge \ i_1 - i_2 < N$$

$$\beta: \quad A_1 = write(A, i_1, i_1) \ \wedge \ y = A_1[i_2 - k] \ \wedge$$
$$y' = A[i_2 - k] \ \wedge \ A'_1 = write(A, i_1, i_1)$$
$$\longrightarrow$$
$$A_1 = A'_1 \ \wedge \ y = y'$$

$$\phi: \quad k - i_2 + i_1 \neq 0$$

$$\phi': \quad k \geq 0 \ \vee \ k \leq -N.$$

The second example is an interchange example given in Fig. 5.3, where the inner and outer loops in the source are exchanged in the target. Observe that this transformation is legal when the leftmost non-zero value in the dependence vector tuple (p, q) has the same sign ($pq \geq 0$) before and after interchange, or there is no loop-carried dependence ($|p| \geq N \ \vee \ |q| \geq |M|$). The following table gives the logical formulas for α, β, ϕ, and ϕ'. The result shows that the interchange is valid when

$$(p \geq 0 \ \vee \ p \leq -N \ \vee \ q \leq 0 \ \vee \ q \geq M) \ \wedge$$

$$(p \leq 0 \ \vee \ p \geq N \ \vee \ q \geq 0 \ \vee \ q \leq -M),$$

which is consistent with our observation.

```
for i = 1 to N                    for j = 1 to M
  for j = 1 to M          ⟹        for i = 1 to N
    A[i, j] = A[i-p, j-q] +1;         A[i, j] = A[i-p, j-q] +1;
```

Figure 5.3: An interchange example

$$
\begin{aligned}
\alpha\colon\ & i_1 - i_2 < 0 \ \wedge\ i_1 - i_2 > -N \ \wedge \\
& j_1 - j_2 > 0 \ \wedge\ j_1 - j_2 < N \\
\beta\colon\ & A_1 = write(A, (i_1, j_1), A[i_1 - p, j_1 - q] + 1) \ \wedge \\
& A_2 = write(A_1, (i_2, j_2), A_1[i_2 - p, j_2 - q] + 1) \ \wedge \\
& A'_1 = write(A, (i_2, j_2), A[i_2 - p, j_2 - q] + 1) \ \wedge \\
& A'_2 = write(A'_1, (i_1, j_1), A'_1[i_1 - p, j_1 - q] + 1) \\
& \qquad\qquad \longrightarrow \\
& \qquad\qquad A_2 = A'_2 \\
\phi\colon\ & (-i_2 + i_1 + p \neq 0 \ \vee\ -j_2 + j_1 + q \neq 0) \ \wedge \\
& (i_2 - i_1 + p \neq 0 \ \vee\ j_2 - j_1 + q \neq 0) \\
\phi'\colon\ & (p \geq 0 \ \vee\ p \leq -N \ \vee\ q \leq 0 \ \vee\ q \geq M) \ \wedge \\
& (p \leq 0 \ \vee\ p \geq N \ \vee\ q \geq 0 \ \vee\ q \leq -M).
\end{aligned}
$$

The third example is the reversal example in Fig. 5.4, where the iteration order of the loop is reversed in the target. Observe that this transformation is legal when there is no loop-carried dependence ($k = 0 \ \vee\ |k| \geq N$). The

following table gives the logical formulas for α, β, ϕ, and ϕ'. The result shows that here reversal is valid only when

$$k = 0 \ \vee \ k \geq N \ \vee \ k \leq -N,$$

which is consistent with our observation.

```
for i = 1 to N                    ⟹    for i = N to 1
    A[i] = A[i-k] + 1;                      A[i] = A[i-k] + 1;
```

Figure 5.4: A reversal example

$$\begin{aligned}
\alpha: \quad & i_1 - i_2 < 0 \ \wedge \ i_1 - i_2 > -N \\
\beta: \quad & A_1 = write(A, i_1, A[i_1 - k] + 1) \ \wedge \\
& A_2 = write(A_1, i_2, A_1[i_2 - k] + 1) \ \wedge \\
& A'_1 = write(A, i_2, A[i_2 - k] + 1) \ \wedge \\
& A'_2 = write(A'_1, i_1, A'_1[i_1 - k] + 1) \\
& \qquad \longrightarrow \\
& \qquad A_2 = A'_2 \\
\phi: \quad & k + i_2 - i_1 \neq 0 \ \wedge \ k - i_2 + i_1 \neq 0 \\
\phi': \quad & k = 0 \ \vee \ k \geq N \ \vee \ k \leq -N.
\end{aligned}$$

The fourth example is a tiling example given in Fig. 5.5, where the two-

level loop is decomposed into a four-level loop such that the iteration space in the source is traversed in tiles of size 10×10 in the target. Observe that this transformation is legal when interchanging is legal ($k \geq 0$) or there is no dependence between the elements in the tiles of the same row ($|k| \geq 10$). The following table gives the logical formulas for α, β, ϕ, and ϕ', where ti_1, ti_2 are the row tile numbers, tj_1, tj_2 are the column tile numbers, and ri_1, ri_2, rj_1, rj_2 are the coordinates within the tiles. The result shows that the tiling is valid when

$$k \geq 0 \ \vee \ k \leq -10,$$

which is consistent with our observation.

```
                          for tilei = 1 to N by 10
for i = 1 to N              for tilej = 1 to M by 10
  for j = 1 to M      ⟹      for i = tilei to Min(N, tilei+9)
    A[i,j] = A[i-k,j-1]+1;    for j =  tilej to Min(M, tilej+9)
                                A[i,j] = A[i-k,j-1] + 1;
```

Figure 5.5: A tiling example

$$\alpha:\ ri_1 - ri_2 < 0\ \wedge\ ri_1 - ri_2 > -10\ \wedge$$

$$rj_1 - rj_2 < 10\ \wedge\ rj_1 - rj_2 > -10\ \wedge$$

$$ti_1 - ti_2 = 0\ \wedge\ tj_1 - tj_2 > 0$$

$$\beta:\ i_1 = 10 * ti_1 + ri_1\ \wedge\ i_2 = 10 * ti_2 + ri_2\ \wedge$$

$$j_1 = 10 * tj_1 + rj_1\ \wedge\ j_2 = 10 * tj_2 + rj_2\ \wedge$$

$$A_1 = write(A, (i_1, j_1), A[i_1 - k, j_1 - 1] + 1)\ \wedge$$

$$A_2 = write(A_1, (i_2, j_2), A_1[i_2 - k, j_2 - 1] + 1)\ \wedge$$

$$A'_1 = write(A, (i_2, j_2), A[i_2 - k, j_2 - 1] + 1)\ \wedge$$

$$A'_2 = write(A'_1, (i_1, j_1), A'_1[i_1 - k, j_1 - 1] + 1)$$

$$\longrightarrow$$

$$A_2 = A'_2$$

$$\phi:\ -ri_2 + ri_1 - k \neq 0$$

$$\phi':\ k \geq 0\ \vee\ k \leq -10$$

5.4 Summary

This chapter proposed an algorithm to generate the conditions which make the loop transformations valid, based on the new permutation rule described in Chapter 3, with the help of an automatic theorem prover CVC Lite. These conditions can be inserted as run-time tests for speculative optimizations. In

this chapter, we also showed the results of implementing the algorithm to generate the run-time tests for speculative loop optimizations by CVC Lite.

In our work, we have established the theory and algorithms for a compiler to generate, or a validator to validate, speculative loop optimizations. While the direction of this work is promising, we have not yet implemented our theory in a working system (although the previous rule, PERMUTE, has been implemented). Though we believe that introducing CVC Lite into the compiler or inserting run-time tests into the executable code should not have a significant overhead, we still need to do more extensive experimentation to obtain convincing performance results.

Chapter 6

Conclusion

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.

I have made the following contributions in my thesis work on translation validation: First, I proposed an improved permutation rule INV-PERMUTE for loop optimizations considering the initial condition and invariant conditions of the loops. Second, I proposed a rule REDUCE for loop reduction. Third, I proposed a generalized validate rule GEN-VALIDATE for structure preserving optimizations, where the control points can be chosen more flexibly so

that some optimizations related to nested loops can be validated. Fourth, I implemented the loop part of TVOC, and improved the tool to analyze loop transformations, synthesize a set of intermediate codes and validated combinations of loop optimizations. Fifth, I presented efficient algorithms to derive the run-time test for speculative optimizations.

Our work on TV can handle a significant variety of compiler optimizations including combinations of both loop optimizations and structure-preserving optimizations. Our goal is to integrate our validation tool into the compiler. Thus, the result of compilation would be not only the optimized target program, but also a simple, machine-checkable proof script. We intend to apply the results to optimizing compilers for emerging architectures, including EPIC processors, out-of-order superscalar machines and highly scalable parallel computing systems.

Bibliography

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[ARG99] Pranav Ashar, Anand Raghunathan, Aarti Gupta, Subhrajit Bhattacharya Verification of Scheduling in the Presence of Loops Using Uninterpreted Symbolic Simulation In *IEEE International Conference on Computer Design*, 1999.

[ASU86] A.V. Aho, R. Sethi and J.D. Ullman. Compilers: Principles, Techniques, and Tool. Addison Wesley, 1986.

[Ban88] U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell, Mass., 1988.

[BB04] Clark Barrett and Sergey 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.

[BFG+05] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Lenore Zuck and Amir Pnueli. TVOC: A Translation Validator for Optimizing Compilers. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, July 2005.

[BGZ03] Clark Barrett, Benjamin Goldberg, and Lenore Zuck. Run-Time Validation of Speculative Optimizations using CVC. In *Oleg Sokolsky and Mahesh Viswanathan, editors, Third International Workshop on Run-time Verification (RV)*, pages 87-105, July 2003.

[BS90] H. Berryman and J. Saltz. A manual for PARTI runtime primitives. In *Interim Report 90-13, ICASE*, 1990.

[DNS03] David Detlefs, Greg Nelson, and James Saxe. Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, Systems Research Center, HP Laboratories, Palo Alto, CA, July 2003.

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

[FORS01] Jean-Christophe Filliatre, Sam Owre, Harald Ruess and N. Shankar. ICS: Integrated Canonizer and Solver. In *Proc. of the 13th Conference on Computer-Aided Verification (CAV01)*, 2001.

[GS99] G. Goos and W. Zimmermann. Verification of Compilers. In *Correct System Design*, volume 1710 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pages 201–230, 1999.

[GZB04] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. In *Proceedings of the Third International Workshop on Compiler Optimization meets Compiler Verificaiton (COCV)*, April 2004.

[HBG04] Ying Hu, Clark Barrett, and Benjamin Goldberg. Theory and algorithms for the generation and validation of speculative loop optimizations. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods*, 2004.

[HBGP05] Ying Hu, Clark Barrett, Benjamin Goldberg and Amir Pnueli. Validating More Loop Optimizations In *Proceedings of the 4th International Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, April 2005.

[HBGZ04] 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*, 2004.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, 12(10):567–580, 1969.

[JCW01] R.D.-C. Ju, S. Chan, and C. Wu. Open Research Compiler (ORC) for the Itanium Processor Family. In Tutorial presented at *Micro 34*, 2001.

[JGS02] C. Jaramillo, R. Gupta, M.L. Soffa. Debugging and testing optimizers through comparison checking In Electronic Notes in Theoretical Computer Science 65 No. 2, 2002.

[LMC03] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically Proving the Correctness of Compiler Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation* (PLDI), 2003.

[MP67] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. In *Proceedings Symposium in Applied Mathematics*, Vol. 19, *Mathematical Aspects of Computer Science* 1967.

[Nec97] G.C. Necula. Proof-carrying code. In *POPL'97*, pages 106–119, 1997.

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

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

[PR98] D. Patel and L. Rauchwerger. Principles of speculative run–time parallelization. In *Proc. of the 11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 1998. Chapel Hill, NC. Also in Lecture Notes in Computer Science, vol. 1656, Springer-Verlag, 1998, pp. 323–338.

[PSS98b] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*, pages 151–166, 1998.

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

Trento, July 2000.

[SBCJ02] K.C. Shashidhar, M. Bruynooghe, F. Catthoor and G. Janssens. Geometric Model Checking: An Automatic Verification Technique for Loop and Data Reuse Transformations. In *Proc. of Compiler Optimization meets Compiler Verificaiton (COCV) 2002*, Electronic Notes in Theoretical Computer Science (ENTCS), volume 65, issue 2.

[ST92] Vivek Sarkar, Radhika Thekkath. A General Framework for Iteration-Reordering Loop Transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, California, pages 175-187, June 1992.

[WL91] M.E. Wolf and M.S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 452-471, Oct. 1991.

[Wolfe95] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.

[ZPFG03] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. Voc: A translation validator for optimizing compilers. In *Journal of Universal Computer Science*, 2003. Preliminary version in *ENTCS*, 65(2), 2002.

[ZPG+05] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of optimized code. In *Formal Methods in Systems Design*, 2005. Preliminary version in *Third Workshop on Runtime Verification (RV),* 2002.