Electronic Notes in Theoretical Computer Science 89 No. 2 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 19 pages

Run-Time Validation of Speculative Optimizations using CVC. [1]

Clark Barrett Benjamin Goldberg Lenore Zuck

Department of Computer Science
New York University
Email:{barrett, goldberg, zuck}@cs.nyu.edu

Abstract

Translation validation is an approach for validating the output of optimizing compilers. Rather than verifying the compiler itself, translation validation mandates that every run of the compiler generate a formal proof that the produced target code is a correct implementation of the source code. Speculative loop optimizations are aggressive optimizations which are only correct under certain conditions which cannot be validated at compile time. We propose using an automatic theorem prover together with the translation validation framework to automatically generate run-time tests for such speculative optimizations. This *run-time validation* approach must not only detect the conditions under which an optimization generates incorrect code, but also provide a way to recover from the optimization without aborting the program or producing an incorrect result. In this paper, we apply the run-time validation technique to a class of speculative reordering transformations and give some initial results of run-time tests generated by the theorem prover CVC.

1 Introduction

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of safety-critical portions of systems. Most verification methods focus on verification of specification with respect to requirements, and high-level code with respect to specification. However, if one is to prove that the high-level specification is correctly implemented in low-level code, one needs to verify the compiler which performs the translations. Verifying the correctness of modern optimizing compilers is challenging because of the complexity and reconfigurability of the target architectures, as well as the sophisticated analysis and optimization algorithms used in the compilers.

[1] This research was supported in by NSF grant CCR-0098299.

©2003 Published by Elsevier Science B. V.

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

Prior work ([PSS98a]) developed a tool for translation validation, CVT, that succeeded in automatically verifying translations involving approximately 10,000 lines of source code in about 10 minutes. The success of CVT critically depends on some simplifying assumptions that restrict the source and target to programs with a single external loop, and assume a very limited set of optimizations.

Our ultimate goal is to develop a methodology for the translation validation of advanced optimizing compilers. Our methods will handle an extensive set of optimizations and can be used to implement fully automatic certifiers for a wide range of compilers, ensuring an extremely high level of confidence in the compiler. This will be particularly valuable in areas such as safety-critical systems and compilation into silicon, where correctness is of paramount concern. We also hope that as a result of this work, future compilers will know how to incorporate appropriate additional outputs into the optimization modules which will facilitate validation. This will lead to a theory of construction of *self-certifying* compilers.

Initial steps towards this goal are described in [ZPFG03]. There, we develop the theory of correct translation. We distinguish between *structure preserving* optimizations, that admit a clear mapping of control points in the target program to corresponding control points in the source program, and *structure modifying* optimizations that admit no such mapping. *Reordering transformations* comprise an important subclass of structure modifying optimizations. Reordering transformations change the order of execution of statements, without altering, deleting, or adding to them. Typical optimizations belonging to this class are *loop interchange*, *loop tiling*, and *loop fusion*.

In some cases, it is impossible to determine at compilation time whether a desired optimization is legal. This is usually because of limited capability to check effectively that syntactically different index expressions refer to the same array location. One possible remedy to this situation is to perform the optimization anyway, adding a run-time check which can determine whether the optimization is safe. If the run-time check fails, the code transfers control to an unoptimized version of the code which completes the computation in a manner which may be slower but is guaranteed to be correct.

In this paper, we focus on leveraging our general theory of correct translation to automatically derive appropriate run-time checks for speculative reordering transformations.

The paper is organized as follows. Following a survey of related work, Section 2 contains an overview of the theory of translation validation. Then, in Section 3, we discuss reordering transformations. We introduce a permutation rule permute-2 for loop fusion and distribution, transformations which are not covered by a similar rule introduced in [ZPFG03,ZPG+03]. In Section 4, we discuss the theorem prover CVC [SBD02], and show how it can be used to verify instances of the permutation rules. Finally, in Section 5, we describe and give examples of how CVC can be used to automatically derive run-time tests for speculative reordering optimizations.

1.1 Related Work

The work here is an extension of the work in [ZPFG03,ZPG+03]. The work in [Nec00] covers some important aspects of our work. For one, it extends the source programs considered from single-loop programs to programs with arbitrarily nested loop structure. An additional important feature is that the method requires no compiler instrumentation at all, and applies various heuristics to recover and identify the optimizations performed and the associated refinement mappings. The main limitation apparent in [Nec00] is that, as is implied by the single proof method described in the report, it can only be applied to structure-preserving optimizations. In contrast, our work can also be applied to structure-modifying optimizations, such as the ones associated with aggressive loop optimizations, which are a major component of optimizations for modern architectures.

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

There is a very large body of literature in the area of optimization and parallelization of loops. For extensive treatments of the subject, see the books by Allen and Kennedy [AK02] and by Wolfe [Wol96]. Most of the literature, including that surveyed by these books, involves approaches for defining notions of dependence, describing optimizations that can be performed once dependence analysis has been performed, and devising compile-time tests to solve the (typically linear) diophantine equations and inequalities that arise when performing dependence analysis in the presence of arrays. These tests include, among others, the GCD test, the Banerjee test [Ban88], the Omega test [Pug92], and various other integer programming techniques such as simplex (see [Fea96] for a description of some of these techniques).

There is also a much smaller body of work on speculative loop optimizations, where a run-time test is generated when the compiler cannot determine if an optimization is safe. Much of this literature concentrates on issues of parallization. Examples of this work include [GN98,RP94,RP95]. In general, papers published in these areas are directed at generating run-time tests to support specific parallization and optimization techniques.

The work we describe in this paper differs from the literature cited above in the following ways:

(i) Rather than developing techniques for generating run-time tests for specific loop optimizations, our run-time tests are automatically generated by a theorem-provier using a general framework (i.e. the permutation rule) for compiler validation. A new loop optimization can be added to a compiler by a compiler writer and, if he or she specifies the permutation that the optimization performs, validation and run-time tests could be automatically generated for the new optimization.

(ii) CVC, the tool we use to prove the verification conditions that we generate as part of the validation process, and the tool that generates the run-time tests for speculative optimization, is a general-purpose automatic theorem prover that has been used successfully in a number of verification contexts. We are currently extending CVC to handle diophatine (i.e. integer) equations in order to improve its effectiveness for validating loop optimizations, but the focus of this work is not to improve the exisiting technology (Omega test, simplex, etc.) for integer programming. Rather, our interest lies in the integration of the compile-time validation process and the generation of run-time tests to support speculative optimizations via the support of a general-purpose theorem prover.

2 Translation Validation of Optimizing Compilers

We outline the general strategy for validation of optimizing compilers and describe the theory of validation of structure preserving optimizations. A more detailed description is in [ZPFG03].

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

The intermediate representation consists of a set of *basic blocks*. Each basic block is a sequence of statements that contain no branches. The control structure can be represented by a *flow graph*, a graph representation in which each node represents a basic block, and the edges represent possible flows of control from one basic block to another.

2.1 Transition Systems

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

- V a set of *state variables*,
- $\mathcal{O} \subseteq V$ a set of *observable variables*,
- Θ an *initial condition* characterizing the initial states of the system, and
- ρ a *transition relation*, relating a state to its possible successors.

The variables are typed, and a *state* of a TS is a type-consistent interpretation of the variables. For a state s and a variable $x \in V$, we denote by $s[x]$ the value that s assigns to x. The transition relation refers to both unprimed and primed versions of the variables, where the primed versions refer to the values of the variables in the successor states, while unprimed versions of variables refer to their value in the pre-transition state. Thus, e.g., the transition relation may include "$y' = y + 1$" to denote that the value of the variable y in the successor state is greater by one than its value in the old (pre-transition) state.

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

A computation of a TS is a maximal finite or infinite sequence of states $\sigma : s_0, s_1, \ldots$, starting with a state that satisfies the initial condition such that every two consecutive states are related by the transition relation.

A transition system $\mathcal{T}$ is called *deterministic* if the observable part of the initial condition uniquely determines the rest of the computation. We restrict our attention to deterministic transition systems and the programs which generate such systems. Thus, to simplify the presentation, we do not consider here programs whose behavior may depend on additional inputs which the program reads throughout the computation. It is straightforward to extend the theory and methods to such intermediate input-driven programs.

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be two TS's, to which we refer as the *source* and *target* TS's, respectively. Two such systems are called *comparable* if there exists a one-to-one correspondence between the observables of P_S and those of P_T. To simplify the notation, we denote by $X \in \mathcal{O}_S$ and $x \in \mathcal{O}_T$ the corresponding observables in the two systems. A source state s is defined to be *compatible* with the target state t, if s and t agree on their observable parts. That is, $s[X] = t[x]$ for every $x \in \mathcal{O}_T$. We say that P_T is a *correct translation* (*refinement*) of P_S if they are comparable and, for every $\sigma_T : t_0, t_1, \ldots$ a computation of P_T and every $\sigma_S : s_0, s_1, \ldots$ a computation of P_S such that s_0 is compatible with t_0, then σ_T is terminating (finite) iff σ_S is and, in the case of termination, their final states are compatible.

Suppose P_S and P_T are comparable TSs. In order to establish that P_T is a correct translation of P_S for the cases that the structure of P_T does not radically differ from the structure of P_S, we designed a proof rule, Validate, which is inspired by the computational induction approach ([Flo67]), originally introduced for proving properties of a single program. Rule Validate provides a proof methodology by which one can prove that one program *refines* another. This is achieved by establishing a *control mapping* from target to source locations, a *data abstraction* mapping from source to target variables, and proving that these abstractions are maintained along basic execution paths of the target program. The proof rule, its soundness, and examples of its applications appear in [ZPFG03,ZPG+03].

3 Validating Loop Reordering Transformations

A *reordering transformation* is a program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statement [AK02]. Reordering transformations cover many of the loop transformations, including fusion, distribution, interchange, tiling, unrolling, and reordering of statements within a loop body.

Because loop transformations are not structure preserving, they cannot be verified directly using rule Validate. In this section we review the reordering loop transformations, and describe some proof rules to validate these transformations.

3.1 Overview of Reordering Loop Transformations

Consider a loop of the form described in Fig. 1.

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

Fig. 1. A General Loop

Equivalently, we can write such a loop in the form **for** $\boldsymbol{i} \in \mathcal{I}$ **by** $\prec_{\mathcal{I}}$ **do** $\mathtt{B}(\boldsymbol{i})$ where $\boldsymbol{i} = (i_1, \ldots, i_m)$ is the list of nested loop indices, and $\mathcal{I}$ is the set of the values assumed by $\boldsymbol{i}$ 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. 1, $\mathcal{I}$ is

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

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. 1, this ordering is the lexicographic order on $\mathcal{I}$.

In general, a loop transformation has the following form:

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

In such a transformation, we may possibly change the domain of the loop indices from $\mathcal{I}$ to $\mathcal{J}$, the names of loop indices from $\boldsymbol{i}$ to $\boldsymbol{j}$, and possibly introduce an additional linear transformation in the loop's body, changing it from the source $\mathtt{B}(\boldsymbol{i})$ to the target body $\mathtt{B}(F(\boldsymbol{j}))$. An example of such a transformation is *loop reversal* which can be described as

$$\textbf{for } i = 1 \textbf{ to } N \textbf{ do } \mathtt{B}(i) \quad \Longrightarrow \quad \textbf{for } j = N \textbf{ to } 1 \textbf{ do } \mathtt{B}(j)$$

For this example, $\mathcal{I} = \mathcal{J} = [1..N]$, the transformation F is the identity, and the two orders are given by $i_1 \prec_{\mathcal{I}} i_2 \longleftrightarrow i_1 < i_2$ and $j_1 \prec_{\mathcal{J}} j_2 \longleftrightarrow j_1 > j_2$, respectively.

Since we expect the source and target programs to execute the same instances of the loop's body (possibly in a different order), we should guarantee that the mapping $F : \mathcal{J} \mapsto \mathcal{I}$ is a bijection from $\mathcal{J}$ to $\mathcal{I}$, i.e. a 1-1 onto mapping. Often, this guarantee can be ensured by displaying the inverse mapping $F^{-1} : \mathcal{I} \mapsto \mathcal{J}$, which for every value of $\boldsymbol{i} \in \mathcal{I}$ provides a unique value of $F^{-1}(i) \in \mathcal{J}$.

Some common examples of transformations which fall into the class considered here are presented in Fig. 2 and Fig. 3. For each transformation, we describe the source loop, target loop, set of loop control variables for source ($\mathcal{I}$) and target ($\mathcal{J}$), their ordering ($\prec_{\mathcal{I}}$ and $\prec_{\mathcal{I}}$), and the bijection $F\colon \mathcal{J} \mapsto \mathcal{I}$. For tiling we assume that c divides m and d divides n.

There are two requirements we wish to establish in order to justify the transformation described in (1).

(i) The mapping F is a bijection from $\mathcal{J}$ onto $\mathcal{I}$. That is, F establishes a 1-1 correspondence between elements of $\mathcal{J}$ and the elements of $\mathcal{I}$.

(ii) For every $\boldsymbol{i}_1 \prec_{\mathcal{I}} \boldsymbol{i}_2$ such that $F^{-1}(\boldsymbol{i}_2) \prec_{\mathcal{J}} F^{-1}(\boldsymbol{i}_1)$, we require that $\mathtt{B}(\boldsymbol{i}_1); \mathtt{B}(\boldsymbol{i}_2) \sim \mathtt{B}(\boldsymbol{i}_2); \mathtt{B}(\boldsymbol{i}_1)$. This requirement is based on the observation that in the source computation, $\mathtt{B}(\boldsymbol{i}_1)$ is executed before $\mathtt{B}(\boldsymbol{i}_2)$ while the corresponding $\mathtt{B}(\boldsymbol{i}_1)$ is executed after $\mathtt{B}(\boldsymbol{i}_2)$ in the target computation. The overall result is the same only if these permutation relations hold between pairs of iterations whose order of execution is reversed in the translation from source to target.

	Interchange	**Skewing**

Source	`for` $\mathtt{i_1 = 1,m}$ `do`
	 `for` $\mathtt{i_2 = 1,n}$ `do`
	 $\mathtt{B}(i_1, i_2)$
		`for` $\mathtt{i_1 = 1,m}$ `do`
		 `for` $\mathtt{i_2 = 1,n}$ `do`
		 $\mathtt{B}(i_1, i_2)$

Target	`for` $\mathtt{j_1 = 1,n}$ `do`
	 `for` $\mathtt{j_2 = 1,m}$ `do`
	 $\mathtt{B}(j_2, j_1)$
		`for` $\mathtt{j_1 = 1,m}$ `do`
		 `for` $\mathtt{j_2 = j_1+1, j_1+n}$ `do`
		 $\mathtt{B}(j_2, j_1 - j2)$

$\mathcal{I}$	$\{1,\ldots,m\} \times \{1,\ldots,n\}$	$\{1,\ldots,m\} \times \{1,\ldots,n\}$
$\mathcal{J}$	$\{1,\ldots,n\} \times \{1,\ldots,m\}$	$\{(j_1,j_2) : 1 \le j_1 \le m \ \wedge$ $j_1 + 1 \le j_2 \le j_1 + n\}$
$\boldsymbol{i} \prec_{\mathcal{I}} \boldsymbol{i'}$	$\boldsymbol{i} <_{\text{lex}} \boldsymbol{i'}$	$\boldsymbol{i} <_{\text{lex}} \boldsymbol{i'}$
$\boldsymbol{j} \prec_{\mathcal{J}} \boldsymbol{j'}$	$\boldsymbol{j} <_{\text{lex}} \boldsymbol{j'}$	$\boldsymbol{j} <_{\text{lex}} \boldsymbol{j'}$
$F(\boldsymbol{j})$	(j_2, j_1)	$(j_1, j_2 - j_1)$
$F^{-1}(\boldsymbol{i})$	(i_2, i_1)	$(i_1, i_1 + i_2)$

Fig. 2. Some Loop Transformations

	Reversal	**Tiling**

Source	`for` $\mathtt{i = 1,n}$ `do`
	 $\mathtt{B}(i)$
		`for` $\mathtt{i_1 = 1,m}$ `do`
		 `for` $\mathtt{i_2 = 1,n}$ `do`
		 $\mathtt{B}(i_1, i_2)$

Target	`for` $\mathtt{j = n,1}$ `do`
	 $\mathtt{B}(j)$
		`for` $\mathtt{j_1 = 1,m}$ `by c`
		 `for` $\mathtt{j_2 = 1,n}$ `by d`
		 `for` $\mathtt{j_3 = j_1, j_1 + c - 1}$
		 `for` $\mathtt{j_4 = j_2, j_2 + d - 1}$
		 $\mathtt{B}(j_3, j_4)$

$\mathcal{I}$	$\{1,\ldots,n\}$	$\{1,\ldots,m\} \times \{1,\ldots,n\}$
$\mathcal{J}$	$\{1..n\}$	$\{(j_1,j_2,j_3,j_3) : 1 \le j_1 \le m \ j_1 \equiv 1 \bmod c \ \wedge$ $1 \le j_2 \le n \ j_2 \equiv 1 \bmod d \ \wedge$ $j_1 \le j_3 < j_1 + c \ \wedge$ $j_2 \le j_4 < j_2 + d\}$
$\boldsymbol{i} \prec_{\mathcal{I}} \boldsymbol{i'}$	$i < i'$	$\boldsymbol{i} <_{\text{lex}} \boldsymbol{i'}$
$\boldsymbol{j} \prec_{\mathcal{J}} \boldsymbol{j'}$	$j > j'$	$\boldsymbol{j} <_{\text{lex}} \boldsymbol{j'}$
$F(\boldsymbol{j})$	j	(j_3, j_4)
$F^{-1}(\boldsymbol{i})$	i	$(c\lfloor\frac{i_1-1}{c}\rfloor + 1, d\lfloor\frac{i_2-1}{d}\rfloor + 1, i_1, i_2)$

Fig. 3. Some Loop Transformations

Our main rule for dealing with loop transformation is Rule PERMUTE presented in Fig. 4. The $\sim$ relation between two programs denotes that the two programs translate one another, i.e., that if each is run from an arbitrary state, the resulting states will be equivalent.

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

Fig. 4. Permutation Rule PERMUTE for reordering transformations

3.2 A Permutation Rule for Loop Fusion and Loop Distribution

The PERMUTE rule given above, does not apply to loop transformations which involve multiple loops that are not nested. Consider the following loop transformation:

$$\begin{array}{l} \textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do} \\ \quad \mathtt{B}_1(i) \\ \textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do} \\ \quad \mathtt{B}_2(i) \end{array} \quad \Longrightarrow \quad \begin{array}{l} \textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do} \\ \quad \mathtt{B}_1(i) \\ \quad \mathtt{B}_2(i) \end{array}$$

This transformation is known as *loop fusion* (the inverse operation is *loop distribution*). Although rule PERMUTE does not apply, the transformation is still a simple permutation of blocks of code. In particular, suppose $\mathcal{I} = \{1 \ldots N\}$. Then loop fusion transforms the sequence

$$\mathtt{B}_1(1); \mathtt{B}_1(2); \ldots; \mathtt{B}_1(N); \mathtt{B}_2(1); \mathtt{B}_2(2); \ldots; \mathtt{B}_2(N)$$

to

$$\mathtt{B}_1(1); \mathtt{B}_2(1); \mathtt{B}_1(2); \mathtt{B}_2(2); \ldots; \mathtt{B}_1(N); \mathtt{B}_2(N)$$

It is easy to see that this transformation preserves equivalence if $\mathtt{B}_2(i)$ is interchangeable with $\mathtt{B}_1(j)$ whenever $i < j$. The rule for loop fusion is formalized in rule PERMUTE-2, shown in Fig. 5.

Since loop distribution is just the inverse of loop fusion, the same rule can be used to verify loop distribution simply by switching the source and the target programs.

$$\frac{\forall \boldsymbol{i}_1, \boldsymbol{i}_2 \in \mathcal{I} : \boldsymbol{i}_1 \prec_{\mathcal{I}} \boldsymbol{i}_2 \quad \longrightarrow \quad \mathtt{B}_1(\boldsymbol{i}_2); \mathtt{B}_2(\boldsymbol{i}_1) \ \sim \ \mathtt{B}_2(\boldsymbol{i}_1); \mathtt{B}_1(\boldsymbol{i}_2)}{\begin{array}{l}\textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}_1(\boldsymbol{i}) \\ \textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}_2(\boldsymbol{i})\end{array} \ \sim \ \textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}_1(\boldsymbol{i}); \mathtt{B}_2(\boldsymbol{i})}$$

Fig. 5. Permutation Rule PERMUTE-2

4 Using CVC to Check Program Equivalence

The Cooperating Validity Checker (CVC) is an automatic theorem prover developed at Stanford University [SBD02]. It is able to check the validity of quantifier-free formulas over a relatively rich set of general-purpose first-order theories. It includes, for example, theories of arrays, linear arithmetic, abstract data types, and uninterpreted functions.

As we will describe in this section, CVC can be used to verify that two pieces of code are equivalent. It is easiest to understand with an example. Suppose we have the following pieces of code:

$$\mathtt{B}_1(i) : a[i] := a[i] + 1;$$

$$\mathtt{B}_2(i) : a[i+1] := a[i-1] - 1;$$

Now suppose we wish to show that these two statements are interchangeable, that is $\mathtt{B}_1(i); \mathtt{B}_2(i) \sim \mathtt{B}_2(i); \mathtt{B}_1(i)$. This can be done by using appropriate variables to model the state of the program before and after each statement. These variables are related using the appropriate transition relation for the statement. To represent the result of $\mathtt{B}_1(i); \mathtt{B}_2(i)$, we first declare three variables for each of i and a. We declare the variables for i to be of type `REAL`, indicating that they are real numbers. The type for the a variables is `ARRAY REAL OF REAL` indicating an array which consists of real numbers and may be indexed by real numbers. (This is actually a convenient abstraction of the actual array. The fact that it is infinite is not a problem since any given formula can only talk about a finite number of array indices.) We then use the CVC `ASSERT` command to say what the relationship between these variables is. The CVC input which represents executing $\mathtt{B}_1(i); \mathtt{B}_2(i)$ is as follows.

```
i0, i1, i2 : REAL;
a0, a1, a2 : ARRAY REAL OF REAL;

ASSERT (a1 = a0 WITH [i0] := a0[i0]+1) AND (i1 = i0);
ASSERT (a2 = a1 WITH [i1+1] := a1[i1-1]-1) AND (i2 = i1);
```

The same thing can be done to represent the result of $\mathtt{B}_2(i); \mathtt{B}_1(i)$. We use the same initial variables but must declare two new variables for each of i and a

to represent the state after executing each of these statements. We can then ask CVC using the QUERY command whether the result of these two executions is the same. The remaining CVC input is as follows.

```
i3, i4 : REAL;
a3, a4 : ARRAY REAL OF REAL;

ASSERT (a3 = a0 WITH [i0+1] := a0[i0-1]-1) AND (i3 = i0);
ASSERT (a4 = a3 WITH [i3] := a3[i3]+1) AND (i4 = i3);

QUERY (i2 = i4 AND a2 = a4);
```

When we submit this query to CVC, it responds with "Valid", indicating that interchanging the two statements does indeed produce equivalent results.

Although the approach just described is probably the most straightforward encoding, in practice we use a variation of this encoding which is more concise. First of all, we can use the same name for variables that don't change (such as i). Second, CVC supports a LET construct which can be used to name an intermediate expression for future reference. Finally, instead of querying whether the two arrays are equal, we query whether reading from the arrays at an arbitrary address gives the same result. Although equivalent, this technique allows CVC to give better information when the proof fails. Using these techniques, we can write the same query as follows.

```
i : REAL;
a : ARRAY REAL OF REAL;
arb_addr : REAL;

QUERY
 (LET a1 : ARRAY REAL OF REAL = a WITH [i] := a[i]+1 IN
     a1 WITH [i+1] := a1[i-1]-1)[arb_addr] =
 (LET a1 : ARRAY REAL OF REAL = a WITH [i+1] := a[i-1]-1 IN
     a1 WITH [i] := a1[i]+1)[arb_addr];
```

Not only is this encoding more concise, it is also typically easier for CVC to prove, primarily because there are fewer variables in the formula.

If a CVC query is not valid, CVC reports "Invalid". When this occurs, the CVC WHERE command gives a set of assumptions under which the formula is not valid. For example, suppose we replace $\mathtt{B}_2(i)$ with the statement $a[i+1] := a[i] - 1$. The CVC query becomes

```
i : REAL;
a : ARRAY REAL OF REAL;
arb_addr : REAL;

QUERY
 (LET a1 : ARRAY REAL OF REAL = a WITH [i] := a[i]+1 IN
```

```
    a1 WITH [i+1] := a1[i]-1)[arb_addr] =
(LET a1 : ARRAY REAL OF REAL = a WITH [i+1] := a[i]-1 IN
    a1 WITH [i] := a1[i]+1)[arb_addr];
```

The query is invalid as expected. The "WHERE" command returns the following information:

```
% Active assumptions:
ASSERT ((1 + 1 * i + -1 * arb_addr) = 0);
```

This indicates that in the case when the "arbitrary address" read from the two arrays is equal to $i + 1$, the two arrays may be different. This helps us pinpoint the problem with interchanging the two statements: $a[i + 1]$ may have the incorrect value after executing the transformed code.

CVC has been designed for large and computationally demanding examples and incorporates recent advances in SAT technology to improve its heuristics. Thus, it is easily able to solve the verification conditions discussed in this paper. CVC also has the ability to produce proofs so that results can be confirmed by an independent proof-checker. This ability supports an ultimate vision in which self-certifying compilers provide a proof which accompanies optimized compiled code.

5 Run-time Validation of Speculative Optimizations

This section builds on the work described in [ZPG+03]. There, we gave an overview of *run-time validation of speculative loop optimizations*, that is, using run-time tests to ensure the correctness of loop optimizations when neither the compiler nor a validation tool are able to. This technique is particularly useful when memory aliasing, due to the use of pointers or arrays, inhibits the static dependence analysis that loop optimizations rely on.

Run-time validation has not only the task of determining when an optimization has generated incorrect code, but also has the task of recovering from the optimization without aborting the program or producing an incorrect result. It is possible in some instances to simply adjust the behavior of the optimized code based on run-time tests, so that correctness is preserved while also maintaining much of the performance benefit of the optimization. In other instances, it is necessary to jump to an unoptimized version of the code.

In [ZPG+03], we showed how run-time tests can enable certain speculative versions of the optimizations discussed in Section 3. However, the run-time tests presented there were constructed manually. We were interested in whether we could use CVC to *automatically* derive and generate appropriate run-time tests for speculative optimizations. This section presents initial results from this effort.

5.1 Strategy

Suppose the compiler wishes to validate a particular loop optimization. As described in Section 4, CVC can be used to check certain kinds of formulas including those describing the equivalence of two blocks of code. In most cases, the conditions listed in rules PERMUTE and PERMUTE-2 can be expressed directly in the input language of CVC.

If CVC reports that the conditions are valid, then the loop optimization is known to preserve the behavior of the program in all contexts. However, if CVC reports that the conditions are not valid, it may still be the case that for many common contexts and cases, the transformation is still valid.

Our strategy for speculative optimization is to use the counter-examples produced by CVC to create run-time tests which guarantee that if the optimized code is used, the result will be correct. We begin by letting ϕ be the (invalid) verification condition for the optimization we wish to perform speculatively.

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

When we are done with the loop, we have a set ψ of conditions under which the transformation is valid. As we will see, some additional work is required to obtain a workable solution from the naive algorithm shown above.

5.2 Interchange Example

Consider the following loop interchange transformation:

```
for i = 1 to M                          for j = 1 to N
   for j = 1 to N                          for i = 1 to M
      k = 10 - j              ==>             k = 10 - j
      A[i, j] = A[i-1, j-k] + C               A[i, j] = A[i-1, j-k] + C
```

The benefits to this loop interchange are 1) the computation of `k` can be moved out of the inner loop and 2) in a language with column-major arrays (such as Fortran), the transformed loop has better locality.

For simplicity, we will only consider the reordering of the array operations, since the correctness of the value of k is trivial. We will use (i_1, j_1) and (i_2, j_2) as two pairs of loop indices. Then, using rule PERMUTE, our initial formula ϕ is:

$$((i_1, j_1) <_{\text{lex}} (i_2, j_2) \quad \wedge \quad (j_2, i_2) <_{\text{lex}} (j_1, i_1)) \quad \longrightarrow$$
$$\mathtt{B}(i_1, j_1); \mathtt{B}(i_2, j_2) \;\sim\; \mathtt{B}(i_2, j_2); \mathtt{B}(i_1, j_1),$$

where B(i,j) is the statement `A[i, j] = A[i-1, j-k] + C`. The input for CVC is:

```
i1, j1, i2, j2, k, C, arb_addr : REAL;
a : ARRAY REAL OF ARRAY REAL OF REAL;

QUERY
((i1 < i2 OR (i1 = i2 AND j1 < j2)) AND
 (j2 < j1 OR (j2 = j1 AND i2 < i1))) =>

((LET a1 : ARRAY REAL OF ARRAY REAL OF REAL =
    a WITH [i1][j1] := a[i1-1][j1-k]+C IN
    a1 WITH [i2][j2] := a1[i2-1][j2-k]+C)[arb_addr] =

 (LET a1 : ARRAY REAL OF ARRAY REAL OF REAL =
    a WITH [i2][j2] := a[i2-1][j2-k]+C IN
    a1 WITH [i1][j1] := a1[i1-1][j1-k]+C)[arb_addr]);
```

This query is invalid, as expected. According to the algorithm given above, we next must select a formula from the counter-example generated by CVC. Among the assumptions returned is the formula $k < 0$. Assuming we are able to automatically select this formula (we will discuss selection criteria below), we negate it and add it as a hypothesis to the query. When we rerun the query, CVC reports "Valid".

Thus, we were able to determine automatically that $\neg(k < 0)$ is sufficient for the correctness of this loop. As described in [ZPG+03], we can then use this result to produce the following correct code. Note that in this example, it is possible to spend part of the computation in the optimized loop, and the rest in the unoptimized loop.

```
for j = 1 to N
    k = 10 - j
    if (k < 0)  goto escape_code
    for i = 1 to M
        A[i, j] = A[i-1, j-k] + C

...

escape_code:

for ii = 1 to M
    for jj = j to N
        k = 10 - jj
        A[ii, jj] = A[ii-1, jj-k] + C
```

Selecting a Formula from the Counter-Example
The only difficult part about automatically generating the run-time test for the example above is choosing a formula from the counter-example returned from CVC.

In order to motivate the selection heuristics, observe that formulas which only relate one of the i variables to one of the j variables cannot contribute to a run-time test. This is because they represent the loop variables at two different points in time and thus cannot be captured by a test at a single point in time. Similarly, some of the other formulas in the counter-example primarily involve constructs which do not correspond to things that can actually be tested at run-time. These include formulas about the introduced variable `arb_addr` or internal CVC constructs. Finally, some formulas in the counter-example are (positive) atomic formulas, and others are negated. The positive formulas tend to express much more information than the negated formulas.

Our heuristic is to choose a positive formula which constrains at least one of the *testable* variables in the formula, where a testable variable is something other than the loop variables, `arb_addr`, and the internal CVC constructs. In the example above, only A, k, and C are testable, and the only formula from the counterexample which constrains one of these is the formula $k > 0$.

5.3 Fusion Example

We now turn to a more sophisticated example. Suppose we have the following procedure which copies N elements from the array pointed to by r to the array pointed to by p.

```
copy(p, r, N)
begin
  for i = 0 to N-1 do
    *(p+i) = *(r+i)
end
```

Now suppose that somewhere in the program two calls are made to this procedure as follows.

```
...
copy(p, r, N);
copy(q, r, N);
...
```

A small procedure like `copy` is likely to be inlined. After inlining, we have the following code.

```
  for i = 0 to N-1 do
    *(p+i) = *(r+i)
  for i = 0 to N-1 do
    *(q+i) = *(r+i)
```

This is a perfect candidate for loop fusion. The fused loop looks like this:

```
for i = 0 to N-1 do
  *(p+i) = *(r+i)
  *(q+i) = *(r+i)
```

This is a highly desirable optimization because it means that each value in the array r only needs to be accessed once instead of twice. Given the number of cycles required to access memory or even the cache (assuming the value is still in the cache the second time), this optimization may result in a significant increase in performance. Unfortunately, today's compilers do not make this optimization because of the danger that the arrays pointed to by p, q, and r may overlap in some way. However, with an appropriate run-time test, we can enable the optimized code.

In order to use CVC to derive the run-time test, we use one big array to model all of memory. The pointers p, q, and r simply become indexes into this array. Now, using rule PERMUTE-2, we obtain the following verification condition.

$$i_1 < i_2 \rightarrow \mathtt{B}_1(i_2); \mathtt{B}_2(i_1) \sim \mathtt{B}_2(i_1); \mathtt{B}_1(i_2),$$

where $\mathtt{B}_1(i)$ is `*(p+i) = *(r+i)` and $\mathtt{B}_2(i)$ is `*(q+i) = *(r+i)`. This verification condition can be expressed in the input language of CVC as follows.

```
p, q, r : REAL;
i1, i2, arb_addr : REAL;
M : ARRAY REAL OF REAL;

QUERY
(i1 < i2) =>

((LET M1 : ARRAY REAL OF REAL =
    M WITH [q+i1] := M[r+i1] IN
    M1 WITH [p+i2] := M1[r+i2])[arb_addr] =

 (LET M1 : ARRAY REAL OF REAL =
    M WITH [p+i2] := M[r+i2] IN
    M1 WITH [q+i1] := M1[r+i1])[arb_addr]);
```

Running this query in CVC produces the following counter-example.

```
ASSERT ((0 + 1 * p + 1 * i2 + -1 * arb_addr) = 0);
ASSERT ((0 + 1 * q + -1 * r + 1 * i1 + -1 * i2) = 0);
ASSERT (0 + 1 * i1 + -1 * i2) < 0;
ASSERT (((0 + 1 * r + 1 * i2 + -1 * arb_addr) = 0) = FALSE);
ASSERT ((M[(0 + 1 * r + 1 * i2)] = M[(0 + 1 * r + 1 * i1)]) =
        FALSE);
```

Only the first two assertions are positive formulas involving at least one testable variable. However, since the first formula only has one testable variable, it is not really expressing a constraint on that variable. Thus, we pick the second formula which states $q - r = i_2 - i_1$. Although this formula includes the non-testable variables i_1 and i_2, we will be able to eliminate them eventually (see below).

After adding the negation of the second assertion as a hypothesis, we get another similar counter-example. Again, we choose the second formula which this time is $q - p = i_2 - i_1$. Running the loop one more time yields another counter-example and another assertion: $r - p = i_2 - i_1$. After adding this hypothesis, the query is valid.

Thus, under the conditions $q - r \neq i_2 - i_1 \wedge q - p \neq i_2 - i_1 \wedge r - p \neq i_2 - i_1$, the transformation is valid. In order to derive a run-time test from this, notice that we have bounds on i_1 and i_2, so the value of $i_2 - i_1$ is always between $-N + 1$ and $N - 1$ inclusive. In fact, because we are using the permute-2 rule, we can also assume that $i_1 < i_2$, so $i_2 - i_1$ must be between 1 and $N - 1$. Thus, we can replace each instance of $\alpha \neq i_2 - i_1$ with $\alpha \leq 0 \vee \alpha \geq N$ to get our final run-time test:

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

After making this replacement, we can again run CVC to check that the query is still valid under the run-time test condition. We must also add conditions for the bounds on i_1 and i_2: $0 \leq i_1 \leq N - 1$ and $0 \leq i_2 \leq N - 1$. As expected, CVC reports that the formula is valid. The final code with the run-time test inserted is:

```
if ((q-r ≤ 0 OR q-r ≥ N) AND
    (q-p ≤ 0 OR q-p ≥ N) AND
    (r-p ≤ 0 OR r-p ≥ N)) begin
  for i = 0 to N-1 do
    *(p+i) = *(r+i)
    *(q+i) = *(r+i)
end
else begin
  for i = 0 to N-1 do
    *(p+i) = *(r+i)
  for i = 0 to N-1 do
    *(q+i) = *(r+i)
end
```

Except for small values of N, the transformed code almost certainly runs faster than the original code.

6 Conclusion

In this paper, we began by reviewing the translation validation approach and introducing a new permutaion rule for loop fusion and loop distribution. We then showed how to use CVC to check the generated verification conditions.

For transformations that cannot be checked at compile-time, we propose using CVC to help generate run-time tests. These tests check one or more conditions which have been proven to guarantee the correctness of the optimized code. If the tests fail, an unoptimized version of the code is run instead.

While the results of this paper are encouraging as a feasibility study in automatically generating verification conditions, there is still much to be understood about the process. The examples given in Section 5 happen to work with our heuristics, but there are other examples for which the current approach is inadequate. We hope that this will be the beginning of an effort to use automatic methods such as those proposed here to generate run-time tests for aggressive speculative compiler optimizations.

References

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

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

[Fea96] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, Lecture Notes in Computer Science (LNCS), Vol. 1132, Springer-Verlag, 1996.

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

[GN98] M. Gupta and R. Nim. Techniques for speculative run-time parallelization. In *Supercomputing'98.* November 1998.

[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.

[PSS98a] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.

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

[Pug92] W. Pugh Eliminating false dependences using the omega test. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programing Language Design and Implementation (PLDI'92).* July 1992.

[RP94] L. Rauchwerger and D. Padua. The privatizing doall test: a run-time technique for doall loop identification and array privatization. In *Proceedings of the 1994 International Conference on Supercomputing*, July 1994.

[RP95] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reductive parallelization. In *Proceedings of ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*. June 1995.

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

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: a cooperating validity checker. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lect. Notes in Comp. Sci.* Springer-Verlag, pages 500–504, 2002.

[Wol96] M.E. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[ZPFG03] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjaming Goldberg. VOC: a translation validator for optimizing compilers. To appear in *Journal of Universal Computer Science*. Preliminary version in *ENTCS*, 65(2), 2002.

[ZPG+03] Lenore Zuck, Amir Pnueli, Benjaming Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of optimized code. To appear in *Formal Methods in Systems Design*. Preliminary version in *Third Workshop on Runtime Verification (RV), 2002.*