Translation and Run-Time Validation of Loop Tranformations*

Lenore Zuck[1], Amir Pnueli[1,2], Benjamin Goldberg[1], Clark Barrett[1], Yi Fang[1], and Ying Hu[1]

[1] Department of Computer Science, New York University
{zuck,amir,goldberg,barrett,yifang,yinghu}@cs.nyu.edu
[2] Weizmann Institute of Science, Rehovot, Israel

Abstract. This paper presents new approaches to the validation of loop optimizations that compilers use to obtain the highest performance from modern architectures. Rather than verify the compiler, the approach of *translation validation* performs a validation check after every run of the compiler, producing a formal proof that the produced target code is a correct implementation of the source code.
As part of an active and ongoing research project on translation validation, we have previously described approaches for validating optimizations that preserve the loop structure of the code and have presented a simulation-based general technique for validating such optimizations. In this paper, for more aggressive optimizations that alter the loop structure of the code – such distribution, fusion tiling, and interchange – we present a set of *permutation rules* which establish that the transformed code satisfies all the implied data dependencies necessary for the validity of the considered transformation. We describe the necessary extensions to VOC-64 in order to validate these structure-modifying optimizations.
This paper also discusses preliminary work on *run-time validation of speculative loop optimizations*, that involves using run-time tests to ensure the correctness of loop optimizations whose correctness can be guaranteed neither at compile time nor by compiler-validation techniques. Unlike compiler validation, 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. This technique has been applied to several loop optimizations, including loop interchange, loop tiling, and software pipelining and appears to be quite promising.

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.

Formally verifying a full-fledged optimizing compiler, as one would verify any other large program, is not feasible, due to its size, evolution over time, and, possibly, proprietary considerations. *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.

The introduction of new families of microprocessor architectures, such as the EPIC family exemplified by the Intel IA-64 architecture, places an even heavier responsibility on optimizing compilers. Static compile-time dependence analysis and instruction scheduling is required to exploit instruction-level parallelism in order to compete with other architectures, such as the super-scalar class of machines where the hardware determines dependences and reorders instructions at run-time. As a result, a new family of sophisticated optimizations have been developed and incorporated into compilers targeted at EPIC architectures.

* This research was supported in part by NSF grant CCR-0098299, ONR grant N00014-99-1-0131, and the John von Neumann Minerva Center for Verification of Reactive Systems.

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.

Other approaches [Nec00,RM00] considered translation validation of less restricted languages allowing, for example, nested loops. They also considered a more extensive set of optimizations. However, the methods proposed there were restricted to *structure preserving* optimizations, and could not directly deal with more aggressive optimizations such as *loop distribution* and *loop tiling* that are often used in more advanced optimizing compilers.

Our ultimate goal is to develop a methodology for the translation validation of advanced optimizing compilers, with an emphasis on EPIC-targeted compilers and the aggressive optimizations characteristic to such 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 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 into the optimization modules appropriate additional outputs 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 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. We provide a general proof rule for translation validation of these transformations and present a tool (VOC-64) that implements the proof rule on an EPIC compiler (SGI Pro-64). An overview of this work is contained in Section 3.

Structure modifying optimizations are more challenging since they do not guarantee a correspondence between control points in the target and the source. This paper discusses *loop reordering transformations* which comprise an important subclass of structure modifying optimizations. Reordering transformations merely change the order of execution of statements, without altering, deleting, or adding to them. Typical optimizations belonging to this class are *loop distribution*, *loop fusion*, *loop tiling*, and *loop interchange*.

For the validation of these reordering transformations we explore "permutation rules" in Section 2. The permutation rules call for calculating which are the reordered statements, and then proving that the reordering preserves the observable semantics. Application of the permutation rules often entails proving that two (loop-free) code fragments are equivalent, thus, validating structure preserving transformations. We describe in Section 3 our approach of validating structure preserving transformations which is often used when validating reordering transformations.

In some cases, it is impossible to determine at compile 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 runtime check which can determine whether the optimization is safe. If the runtime check fails, control is transferred to an unoptimized version of the code which completes the computation in a manner which may be slower but is guaranteed to be correct. This method is presented in Section 4.

1.1 Related Work

The work here is an extension of the work in [ZPFG03]. In fact, the main of [ZPFG03] and tool that implements it were reworked to accommodate the work reported here, and their performance was enhanced with respect to the global optimizations which they were designed to handle. [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.

The notion of correct translation that appears in [GS99] is similar to ours; however, the work there does not deal with optimizations.

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].

More weakly related are the works reported in [Nec97] and [NL98], which do not purport to establish full correctness of a translation but are only interested in certain "safety" properties. However, the techniques of program analysis described there are very relevant to the automatic generation of refinement mappings and auxiliary invariants. Rival [Riv03] presents a methodology based on abstract-interpretation for certification of assembly code that uses the analysis of the source code and the debugging information.

The work in [Fre02] presents a framework for describing global optimizations by rewrite rules with CTL formulae as side conditions, which allow for generation of correct optimizations, but not for verification of (possibly incorrect) optimizations. The work in [GGB02] proposes a method for deploying optimizing code generation while correct translation between input program and code. They focus on code selection and instruction scheduling for SIMD machines.

Somewhat similar to our approach, in the treatment of structure modifying transformation, is the work in [SBCJ02]. There, static analysis is used to extract geometric model, on which loop data reuse transformations are checked to preserve semantics. preserving equivalence conditions can be checked that . However, this work focuses on source-to-source transformations that are manually applied in embedded system design, while the focus of our work is optimizations that are automatically generated by compilers.

2 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]. It preserves a dependence if it preserves the relative execution order of the source and target of that dependence, and thus preserves the meaning of the program. Reordering transformations cover many of the loop transformations, including fusion, distribution, interchange, tiling, unrolling, and reordering of statements within a loop body.

In this section we review the reordering loop transformation and propose "permutation rules" that the validator may use to deal with these transformations.

2.1 Overview of Reordering Loop Transformations

Consider the generic loop transformation in Fig. 1.

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

Fig. 1. A General Loop

Schematically, we can describe such a loop in the form "$\textbf{for } \vec{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\vec{i})$" where $\vec{i} = (i_1, \ldots, i_m)$ is the list 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. 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 form:

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

In 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 $\vec{i}$ to $\vec{j}$, and possibly introduce an additional linear transformation in the loop's body, changing it from the source $\mathtt{B}(\vec{i})$ to the target body $\mathtt{B}(F(\vec{j}))$.

An example of such a transformation is *loop reversal* which can be described as

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

For this example, $\mathcal{I} = \mathcal{J} = [1..N]$, the transformation F is the identity, and the two orders are given by $i_1 \prec_{\mathcal{I}} i_2 \iff i_1 < i_2$ and $j_1 \prec_{\mathcal{J}} j_2 \iff j_1 > j_2$, 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 defining the inverse mapping $F^{-1} : \mathcal{I} \mapsto \mathcal{J}$, which for every value of $\vec{i} \in \mathcal{I}$ provides a unique value of $F^{-1}(i) \in \mathcal{J}$.

Some common examples of transformations which fall into the class considered here[3] 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 $\prec_{\mathcal{J}}$), and the bijection $F \colon \mathcal{J} \mapsto \mathcal{I}$. For tiling we assume that c divides n and d divides m.

	Interchange	**skewing**

Source	`for i1 = 1,n do`
	`for i2 = 1,m do`
	$\mathtt{B}(i_1, i_2)$
		`for i1 = 1,n do`
		`for i2 = 1,n do`
		$\mathtt{B}(i_1, i_2)$

Target	`for j1 = 1,m do`
	`for j2 = 1,n do`
	$\mathtt{B}(j_2, j_1)$
		`for j1 = 1,n do`
		`for j2 = j1 + 1, j1 + m do`
		$\mathtt{B}(j_1, j_2 - j_1)$

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

Fig. 2. Some Loop Transformations

2.2 Permutation Rules

When applying the translation validation approach to optimizations, we distinguish between two classes of program transformations. One is the *structure preserving* class, that admits a natural mapping between the program states of the source and target after the execution of each block of statements that consists of a small set of instructions. (More formally, the mapping should exist at cutpoint of the control flow graph of the program. We elaborate on this point in the next section.) For such structure preserving optimizations, we apply a Floyd-like simulation based proof method, where verification conditions are constructed whose validity is then checked by a theorem prover.

[3] Technically, skewing is not a reordering transformation, since the loop body code is changed. However, our methodology easily handles the validation of skewing, so we include skewing in this paper

	Reversal	Tiling
Source	`for i = 1,n do` `B(i)`	`for i1 = 1,n do` `for i2 = 1,m do` `B(i1, i2)`
Target	`for j = n,1 do` `B(j)`	`for j1 = 1,n by c do` `for j2 = 1,m by d do` `for j3 = j1, j1 + c − 1 do` `for j4 = j2, j2 + d − 1 do` `B(j3, j4)`
$\mathcal{I}$	$\{1, \ldots, n\}$	$\{1, \ldots, n\} \times \{1, \ldots, m\}$
$\mathcal{J}$	$\{1, \ldots, n\}$	$\{(j_1, j_2, j_3, j_4) : 1 \leq j_1 \leq n \ \wedge \ j_1 \equiv 1 \bmod c \ \wedge \ 1 \leq j_2 \leq m \ \wedge \ j_2 \equiv 1 \bmod d \ \wedge \ j_1 \leq j_3 < j_1 + c \ \wedge \ j_2 \leq j_4 < j_2 + d\}$
$\vec{i} \prec_{\mathcal{I}} \vec{i'}$	$i < i'$	$\vec{i} <_{\text{lex}} \vec{i'}$
$\vec{j} \prec_{\mathcal{J}} \vec{j'}$	$j > j'$	$\vec{j} <_{\text{lex}} \vec{j'}$
$\vec{i} = F(\vec{j})$	$i = j$	$(i_1, i_2) = (j_3, j_4)$
$\vec{j} = F^{-1}(\vec{i})$	$j = i$	$(j_1, j_2, j_3, j_4) = (c\lfloor \frac{i_1 - 1}{c} \rfloor + 1, d\lfloor \frac{i_2 - 1}{d} \rfloor + 1, i_1, i_2)$

Fig. 3. Some Loop Transformations

The second class of program transformations is the *structure modifying* that admits no natural mapping between the states of the source and target programs at each cutpoint. The reordering transformations, which are the topic of this paper, are such structure modifying transformations, since each entrance to a loop is a cutpoint, and transformations that alter the order of executions of the the loop body cannot be handled by the simulation based approach.

In this section we develop a rule called the "permutation rule" for validating reordering transformations and establish the soundness of the permutation rule. Usually, structure-modifying optimizations are applied to small localized sections of the source program, while the rest of the program is only optimized by structure-preserving transformations. Therefore, the general validation of a translation will combine these two techniques.

Assume for now that we have means to establish simulations between two program fragments, P and Q, that are in between two cutpoints of the flow graph. We denote the simulation between them by $P \sim Q$. (Section 3 elaborates how $P \sim Q$ is established.) Note that $\sim$ is reflexive, transitive, and closed under the sequential composition operator ";", i.e., if $P \sim Q$, then $P; R \sim Q; R$.

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

1. 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 $\mathcal{I}$.
2. For every $\vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2$ such that $F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1)$, $\mathtt{B}(\vec{i}_1); \mathtt{B}(\vec{i}_2) \sim \mathtt{B}(\vec{i}_2); \mathtt{B}(\vec{i}_1)$. This requirement is based on the observation that in the source computation, $\mathtt{B}(\vec{i}_1)$ is executed before $\mathtt{B}(\vec{i}_2)$ while the corresponding $\mathtt{B}(\vec{i}_2)$ is executed before $\mathtt{B}(\vec{i}_1)$ in the target computation. The overall results would be the same if these permutation relations hold between pairs of iterations whose order of execution is reversed between source and target.

These requirements are summarized in rule PERMUTE, presented in Fig. 4.

$$\begin{array}{ll}
\text{R1. } \forall \vec{i} \in \mathcal{I} : \exists \vec{j} \in \mathcal{J} : & \vec{i} = F(\vec{j}) \\
\text{R2. } \forall \vec{j}_1 \neq \vec{j}_2 \in \mathcal{J} : & F(\vec{j}_1) \neq F(\vec{j}_2) \\
\text{R3. } \forall \vec{i}_1, \vec{i}_2 \in \mathcal{I} : & \vec{i}_1 \prec_{\mathcal{I}} \vec{i}_2 \wedge F^{-1}(\vec{i}_2) \prec_{\mathcal{J}} F^{-1}(\vec{i}_1) \implies \mathtt{B}(\vec{i}_1); \mathtt{B}(\vec{i}_2) \sim \mathtt{B}(\vec{i}_2); \mathtt{B}(\vec{i}_1) \\
\hline
\multicolumn{2}{c}{\textbf{for } \vec{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\vec{i}) \quad \sim \quad \textbf{for } \vec{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\vec{j}))}
\end{array}$$

Fig. 4. Permutation Rule PERMUTE for reordering transformations

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

Lemma 1 (Soundess of Permute). *Let $\mathcal{I}$ and $\mathcal{J}$ be finite sets of integer vectors 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. If*

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

then

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

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

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

then

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

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

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

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 $\vec{j}_* > \vec{j}_k$, then the inductive step is established. Else, let ℓ be the minimal such that $\vec{j}_* \prec_{\mathcal{J}} \vec{j}_\ell$. It suffices to show that

$$\mathtt{B}(F(\vec{j}_1)); \ldots; \mathtt{B}(F(\vec{j}_{\ell-1})); \mathtt{B}(\vec{j}_*); \mathtt{B}(F(\vec{j}_\ell)); \ldots; \mathtt{B}(F(\vec{j}_k)) \ \sim \ \textbf{for } \vec{j} \in \mathcal{J}_k \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\vec{j}));\ \mathtt{B}(\vec{j}_*)$$

The proof is by induction on $t = k, \ldots, \ell$, where, for each m, since $\vec{j}_* \prec_{\mathcal{J}} \vec{j}_t$, the assumptions of the claim imply that $\mathtt{B}(\vec{j}_t); \mathtt{B}(\vec{j}_*) \sim \mathtt{B}(\vec{j}_*); \mathtt{B}(\vec{j}_t)$, thus $\mathtt{B}(\vec{j}_*)$ can be "bubbled" into its position in between $\mathtt{B}(\vec{j}_\ell)$ and $\mathtt{B}(\vec{j}_{\ell+1})$. □

In order to apply rule PERMUTE to a given case, it is necessary to identify the function F (and F^{-1}) and validate Premises R1–R3 of Rule PERMUTE. The identification of F can be provided to us by the compiler, once it determines which of the relevant loop optimizations it chooses to apply. Of course, the validator will need to verify that F is a bijection and the source and target bodies relate to each other as prescribed by the rule. Alternately, we can develop heuristics which can attempt to identify these functions based on a comparison of the source and target codes.

In most of the applications, the functions F and F^{-1} are linear expressions which can be easily handled by CVC. The only obvious exception is tiling, but then one can express the F's using an existential quantifier, which is still within the scope of CVC.

The techniques presented here only deal with transformations which reorder the execution of the entire loop's body. They can be easily generalized to deal with cases where the loop's body is partitioned into several segments, and each of the segments is moved to a different iteration. Such transformations occur in the case of loop fusion and distribution as well as software pipelining.

Proving the $\sim$ relation may be trickier. In many simple examples, which we present in this section, the loop bodies consist of a single statement involving a linear expression, and then we can use CVC to validate it. In more realistic cases, where the loop transformation is the first step for other (global) optimizations, it may be necessary to establish the verification conditions that imply the validity of the $\sim$ relation in Premise R3. We outline the mechanism of doing that in Section 3.

2.3 Examples

In this section, we present examples that illustrate the use of the permutation function in Fig. 4 for loop interchange, tiling, and skewing. Hopefully, this will clarify for the reader the choice of the sets $\mathcal{I}$ and $\mathcal{J}$, the functions F and F^{-1}, and the relations $\prec_{\mathcal{I}}$ and $\prec_{\mathcal{J}}$ for these optimizations.

Loop Interchange An example of loop interchange is the transformation in Fig. 5. This transformation is useful, for example, in languages (like Fortran) where arrays are column major. Better cache performance is achieved by traversing the columns of the array than traversing the rows (i.e. it is better for the first subscript to be changing faster).

```
for i = 1, N do                               for j = 2, M do
    for j = 2, M do               ==>             for i = 1, N do
        a[i,j] = a[i-1, j-1] + c                      a[i,j] = a[i-1, j-1] + c
```

Fig. 5. Loop Interchange Example

Looking at Fig. 2, we see that for this example:

$$
\begin{array}{rcl}
\mathcal{I} & = & \{(i,j) | 1 \le i \le N, 2 \le j \le M\} \qquad\qquad \mathcal{J} = \{(j,i) | 2 \le j \le M, 1 \le i \le N\} \\
(i_1,j_1) \prec_{\mathcal{I}} (i_2,j_2) & \iff & (i_1 < i_2) \ \vee \ ((i_1 = i_2) \ \wedge \ (j_1 < j_2)) \\
(j_1,i_1) \prec_{\mathcal{J}} (j_2,i_2) & \iff & (j_1 < j_2) \ \vee \ ((j_1 = j_2) \ \wedge \ (i_1 < i_2)) \\
F(j,i) & = & (i,j) \qquad\qquad\qquad\qquad F^{-1}(i,j) = (j,i)
\end{array}
$$

Plugging these terms into premise R3 of the permutation rule of Fig. 4, we generate the following verification condition for CVC to validate:

$$
\begin{array}{l}
\forall (i_1,j_1),(i_2,j_2) \in \mathcal{I} : \\
\qquad (i_1,j_1) \prec_{\mathcal{I}} (i_2,j_2) \ \wedge \ (j_2,i_2) \prec_{\mathcal{I}} (j_1,i_1) \implies \\
\qquad\qquad (\texttt{a[}i_1\texttt{,}j_1\texttt{] = a[}i_1\texttt{-1, }j_1\texttt{-1] + c; a[}i_2\texttt{,}j_2\texttt{] = a[}i_2\texttt{-1, }j_2\texttt{-1] + c}) \ \sim \\
\qquad\qquad\qquad (\texttt{a[}i_2\texttt{,}j_2\texttt{] = a[}i_2\texttt{-1, }j_2\texttt{-1] + c; a[}i_1\texttt{,}j_1\texttt{] = a[}i_1\texttt{-1, }j_1\texttt{-1] + c})
\end{array}
$$

CVC does validate this particular verification condition. If, however, we change the assignment statement in the body of the loop to,

```
a[i,j] = a[i-1, j+1] + c
```

then the validation fails.

Tiling The tiling optimization, also referred to as blocking and unroll-and-jam, modifies the access pattern over arrays in order to improve data cache performance. Typically, repeated traversals across an entire dimension (such as a row) of an array are replaced by repeated traversals across sections (tiles/blocks) within the array where, hopefully, each block is able to fit in cache. The canonical example of tiling is the blocking form of matrix multiplication, as in See Fig. 6, where where each of the matrices a, b, and e are divided into blocks of size $c \times c$, which hopefully fit into the cache. For the purpose of this discussion, we'll assume, in all the examples involving tiling, that the tile size (in this case, `c`) divides the number of iterations of each loop (e.g. N, M, and L) evenly.

```
                                                 for ii = 1,N by c do
                                                     for jj = 1,M by c do
for i = 1, N do                                          for kk = 1,L by c do
    for j = 1,M do                 ==>                       for i = ii, ii+c-1 do
        for k = 1,L do                                           for j = jj,jj+c-1 do
            a[i,j] = a[i,j] + e[i,k] * b[k,j]                        for k = 1,kk+c-1 do
                                                                         a[i,j] = a[i,j] + e[i,k] * b[k,j]
```

Fig. 6. Matrix Multiplication

In the compiler literature, the enabling condition for tiling is often stated as being if the loops are fully permutable (interchangeable). The use of the definitions of $\mathcal{I}$, $\mathcal{J}$, $\prec_{\mathcal{I}}$, $\prec_{\mathcal{J}}$, etc. – appearing in the rightmost column of Fig. 3 – in the permutation rule gives us a less restrictive enabling condition for tiling. It essentially says that even if a loop isn't interchangeable, it can still be tileable if every dependence is either within the same tile (block) spans two rows of tiles.

A simple example of a non-interchangeable, however, tileable loop is in Fig. 7.

```
for i = 6, N do
    for j = 1, M-10 do
        a[i,j] = a[i-5,j+10] + d
```

Fig. 7. A non-interchangeable tileable loop

```
for ii = 6,N by 5 do
    for jj = 1,M-10 by 5 do
        for i = ii, ii+4 do
            for j = jj,jj+4 do
                a[i,j] = a[i-5,j+10] + d
```

Fig. 8. 5×5 Tiling of the loop in Fig. 7

It is tileable as long as the tile size is no greater than 10 (arising from the occurrence of `j+10`). The transformation, for example, of this loop to the loop in Fig. 8 would result in the following definitions:

$$\begin{array}{rcl}
\mathcal{I} & = & \{(i,j) : 1 \le i \le N,\ 1 \le j \le M-10\} \\
\mathcal{J} & = & \{(ii,jj,i,j) : \begin{pmatrix} 6 \le ii \le N \ \wedge\ 1 \le jj \le M-10 \ \wedge\ ii \le i \le ii+4 \ \wedge \\ jj \le j \le jj+4 \ \wedge\ ii \equiv 1 \bmod 5 \ \wedge\ jj \equiv 1 \bmod 5 \end{pmatrix}\} \\
(i_1,j_1) \prec_{\mathcal{I}} (i_2,j_2) & \Longleftrightarrow & (i_1 < i_2) \ \vee\ ((i_1 = i_2) \ \wedge\ (j_1 < j_2)) \\
(ii_1,jj_1,i_1,j_1) \prec_{\mathcal{J}} (ii_2,jj_2,i_2,j_2) & \Longleftrightarrow & (ii_1 < ii_2) \ \vee\ ((ii_1 = ii_2) \ \wedge\ (jj_1 < jj_2)) \ \vee \\
 & & ((ii_1 = ii_2) \ \wedge\ ((jj_1 = jj_2) \ \wedge\ (i_1 < i_2)) \ \vee \\
 & & ((ii_1 = ii_2) \ \wedge\ ((jj_1 = jj_2) \ \wedge\ (i_1 = i_2) \ \wedge\ (j_1 < j_2)) \\
F(ii,jj,i,j) & = & (i,j) \qquad F^{-1}(i,j) = (5\lfloor \frac{i-1}{5} \rfloor + 1, 5\lfloor \frac{j-1}{5} \rfloor + 1, i, j)
\end{array}$$

The verification condition, resulting from premise R3 of the permutation rule, was sent to, and successfully validated, by CVC.

2.4 A Simplified Permutation Rule

In the optimizing compiler literature (see, for example, [?]), a loop reordering transformation is regarded as being safe if it preserves the dependences of the original loop. That is, it preserves the relative order of execution of two accesses to the same location in memory (or register), if at least one of those accesses is a write. The particular dependences are:

- *flow dependence*, where a write to a location is followed by read from the same location,
- *anti-dependence*, where a read from a location is followed by a write to the same location,
- *output dependence*, where a write to a location is followed by another write to same location.

Notice that the permutation rule described in figure Fig. 4 provides a more general statement about the correctness of loop reordering transformations than the rule based on dependence. This is because there are loops that can be reordered without changing its observable behavior, while not preserve dependences. A simple example of of this arises when performing loop interchange is in Fig. 9.

The transformation in Fig. 9 is clearly correct – since 0 is written to each element of the array – but it violates flow dependences. In the original loop, the memory location written to as $a[i,j]$ is later read as $a[i-1,j+1]$. In the transformed loop, the location read as $a[i-1,j+1]$ is later written as $a[i,j]$.

```
for i = 1,n do                                  for j = 1,m do
    for j = 1,m do                       ==>        for i = 1,n do
        a[i,j] = a[i-1,j+1] - a[i-1,j+1]                a[i,j] = a[i-1,j+1] - a[i-1,j+1]
```

Fig. 9. A simple loop interchange

Since compilers don't typically perform the kind of algebraic manipulation that recognizes when a transformation is safe despite dependence violations, we can simplify Rule PERMUTE and simplify the query sent to CVC (as well as the discussion here):

Suppose there is a transformation of a loop, from

for $\vec{i} \in \mathcal{I}$ by $\prec_{\mathcal{I}}$ do
  a[$D(\vec{i})$] = ...
  ... = a[$U(\vec{i})$]

to

for $\vec{j} \in \mathcal{J}$ by $\prec_{\mathcal{J}}$ do
  a[$D(F(\vec{j}))$] = ...
  ... = a[$U(F(\vec{j}))$]

such that:

- D determines the array element being written to (i.e. *defined*) and U determines the array element being read (i.e. *used*).
- F is a bijection from $\mathcal{J}$ to $\mathcal{I}$ (as before).

For this discussion, we'll assume there are no other reads and writes to arrays in the loop – although additional reads and writes are easily handled.

A simplified rule that we use is the following:

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

This more closely reflects the traditional definition of a safe reordering. It states that if, in the original loop, there are two iterations where the loop variable(s) takes on the values $\vec{i}_1$ and $\vec{i}_2$, respectively, such that the iteration corresponding to $\vec{i}_1$ occurs before that corresponding to $\vec{i}_2$ and there is a flow, anti-, or output dependence between them, then the iteration in the transformed loop $\vec{i}_1$ must occur before the iteration corresponding to $\vec{i}_2$.

Using CVC we have proved that, for reversal an interchange transformations for a loop body of the simplified form above (and uninterpreted functions), a transformation that satisfies the simplified rule also satisfies Rule PERMUTE of Fig. 4.

In Section 4 we use the simplified rule in the discussion of *run-time* validation of the optimized code, where, if translation validation fails at compile-time due to incomplete knowledge, then run-time tests are inserted in the program to assure the correctness of the transformations.

3 Translation Validation of Optimizing Compilers

Rule PERMUTE of the previous section calls for establishing a simulation relation $\sim$ between two program fragments. As we mentioned there, establishing this relation automatically for non-trivial code fragments is often challenging. In previous work ([ZPFG03]) we presented the theory and a proof rule, VALIDATE, for validating order preserving transformations. The gist of the proof rule is an automatic construction of *verification conditions* whose validity is checked by a theorem prover. We also developed a tool, VOC-64, that automatically constructs the verification conditions according to VALIDATE and sends them to validation by ICS ([FORS01]).

The original Rule VALIDATE constructed verification conditions that have an existential quantifier on the right hand side of an implication. In addition of being a source of confusion to the readers of our work, it also implied some inefficiency in the validation process. In fact, the original implementation of VOC-64 bypassed this by using heuristics to instantiate the existentially quantified variables.

As part of our work on validating the reordering transformations, and the incorporation of CVC in VOC-64, we realized that, for deterministic programs, Rule VALIDATE can be cast without the existential quantifier, creating verification conditions that can be easily validated by CVC. It is this new, more elegant, version of the rule that we are currently using for validation of both order preserving and reordering transformations.

In this section we describe the formal model we use and the new VALIDATE proof rule. The description of the rule is rather terse, being only slightly different than the similar description in [ZPFG03]. We include an example of the verification conditions obtained by the new rule that was derived by the new version of VOC-64 when applied a series of optimizations performed by the SGI Pro-64 compiler.

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 single-entry single-exit sequence of statements that contains no branches (but in the last statement). 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.

3.1 Transition Systems

In order to discuss the formal semantics of programs, 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. We assume that each transition system has a variable 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 execution path of a program. E.g., consider a basic block consisting of:

```
B0:
     n <- 500
     y <- 0
     w <- 1
     IF !(n >= w) GOTO B2
B1:
```

There are two disjunct in the transition relation associated with this block. The first describes the `B0` to `B1` path, which is $\pi = \mathtt{B0} \ \wedge \ n' = 500 \ \wedge \ y' = 0 \ \wedge \ w' = 1 \ \wedge \ n' \geq w' \ \wedge \ \pi' = \mathtt{B1}$, and the second describes the `B0` to `B2` path, which is $\pi = \mathtt{B0} \ \wedge \ n' = 500 \ \wedge \ y' = 0 \ \wedge \ w' = 1 \ \wedge \ n' < w' \ \wedge \ \pi' = \mathtt{B2}$, where π denotes the program location counter.

The transition relation is then the disjunction of all such generalized transition relations.

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 (i.e., 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.

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.

3.2 Translation Validation of Structure-Preserving Transformations

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be comparable TS's, where P_S is the *source* and P_T is the *target*. In order to establish that P_T is a correct translation of P_S for the cases that the structure of P_T does not radically differ from the structure of P_S, we use a proof rule, VALIDATE, which is inspired by the computational induction approach ([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 variables to (possibly guarded) expressions over the target variables, and proving that these abstractions are maintained along basic execution paths of the target program.

The proof rule, VALIDATE, is presented in Fig. 10. It is essentially the proof rule VALIDATE of [ZPFG03], with some modifications to accommodate easier automatic proofs. After we describe this current version of the rule, we discuss the differences between it and its predecessor, and prove their equivalence.

In the proof rule of Fig. 10, each TS is assumed to have a *cut-point set* CP. This is a set of blocks that includes all initial and terminal blocks, as well as at least one block 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. We denote the source control variable is `PC` and the target's is `pc`; the range of the control variables is obviously the cutpoint set.

For each simple path leading from `Bi` to `Bj`, ρ_{ij} describes the transition relation between blocks `Bi` and `Bj`. Typically, such a transition relation contains the condition which enables this path to be traversed (including the condition that the control variable be equal to i) and the data transformation effected by the path. Note that, when the path from `Bi` to `Bj` passes through blocks that are not in the cut-point set, ρ_{ij} is a compressed transition relation that can be computed by the composition of the intermediate transition relations on the path from `Bi` to `Bj`.

The proof rule 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.

The invariants φ_i in part (2) are program annotations that are expected to hold whenever execution visits block `Bi`. They often can be derived from the data flow analysis carried out by an optimizing compiler. Intuitively, their role is to carry information in between basic blocks that is needed, for example, when constant propagation is code elimination are applied.

1. Establish a *control abstraction* $\kappa\colon \mathsf{CP}_T \to \mathsf{CP}_S$ that maps initial and terminal blocks of the target into the initial and terminal blocks of the source.
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*

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

   which asserts that the source and target are at corresponding 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`, $\Theta_S \ \wedge \ \Theta_T \to \alpha \ \wedge \ \varphi_i$. 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

   $$C_{ij}: \qquad \varphi_i \ \wedge \ \alpha \ \wedge \ \rho^T_{ij} \ \wedge \ (\bigvee_{\pi \in Paths^S} \rho_\pi) \quad \to \quad \alpha' \ \wedge \ \varphi'_j,$$

   where $Paths^S$ is the set of all simple source paths and ρ_π is the transition relation for the simple source path π.
5. Establish the validity of all the generated verification conditions.

Fig. 10. The Proof Rule VALIDATE

The verification conditions assert that at each (target) transition from `Bi` to `Bj`[4], if the assertion φ_i and the data abstraction hold before the transition, and the transition takes place, then after the transition, the data abstraction and the assertion φ_j hold in the new state. Hence, φ_i is used as a hypothesis at the antecedent of the implication C_{ij}. In return, the validator also has to establish that φ_j holds after the transition. Thus, as part of the verification effort, we confirm that the proposed assertions are indeed inductive and hold whenever the corresponding block is visited.

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. The approach promoted here will make sense only if this validation (as well as the preceding steps of the conditions' generation) can be done in a fully automatic manner with no user intervention.

[ZPFG03] contains a discussion, soundness proof, and examples of applications of the rule. As mentioned above, the version of the rule presented here differs from that presented in [ZPFG03]. The main difference is that we have changed the verification condition in (4) to eliminate the existential quantifier. Elimination of the existential quantifier makes it more likely that the generated verification conditions can be checked automatically. The following theorem shows that the rules are equivalent under the assumption that the transition systems are deterministic.

Theorem 1. *The following verification conditions are equivalent:*

$$\varphi_i \ \wedge \ \alpha \ \wedge \ \rho^T_{ij} \quad \to \quad \exists {V_S}' : (\bigvee_{\pi_1 \in Paths(\kappa(i),\kappa(j))} \rho^S_{\pi_1}) \ \wedge \ \alpha' \ \wedge \ \varphi'_j, \tag{2}$$

[4] Recall that we assume that a path described by the transition is simple.

and

$$\varphi_i \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho}_{ij} \ \wedge \ (\bigvee_{\pi_2 \in Paths(\kappa(i))} \overset{S}{\rho}_{\pi_2}) \quad \rightarrow \quad \alpha' \ \wedge \ \varphi'_j, \tag{3}$$

Proof. In one direction, suppose (3) holds and suppose that we have $\varphi_i \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho}_{ij}$. By definition of ρ and α, it follows that $\mathtt{pc} = \kappa(i)$. Without loss of generality, we can assume that at every non-terminal source cut-point, some transition must be taken and that at terminal source cut-points, no transitions are enabled. It then follows that $(\bigvee_{\pi_2 \in Paths(\kappa(i))} \overset{S}{\rho}_{\pi_2})$ holds. Thus, by (3), we have $\alpha' \ \wedge \ \varphi'_j$. But from $\overset{T}{\rho}_{ij} \wedge \alpha'$, $\mathtt{pc}' = \kappa(j)$ follows. We thus have $(\bigvee_{\pi_1 \in Paths(\kappa(i),\kappa(j))} \overset{S}{\rho}_{\pi_1}) \ \wedge \ \alpha' \ \wedge \ \varphi'_j$, and so clearly we also have $\exists {V_S}' \colon (\bigvee_{\pi_1 \in Paths(\kappa(i),\kappa(j))} \overset{S}{\rho}_{\pi_1}) \ \wedge \ \alpha' \ \wedge \ \varphi'_j$.

In the other direction, suppose that (2) holds and suppose that we have $\varphi_i \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho}_{ij} \ \wedge \ (\bigvee_{\pi_2 \in Paths(\kappa(i))} \overset{S}{\rho}_{\pi_2})$. In particular, one of the transitions $\overset{S}{\rho}_{\pi_2}$ is true. By (2), we have $\exists {V_S}' \colon (\bigvee_{\pi_1 \in Paths(\kappa(i),\kappa(j))} \overset{S}{\rho}_{\pi_1}) \ \wedge \ \alpha' \ \wedge \ \varphi'_j$. Since φ'_j only contains target variables, we can immediately conclude that φ'_j holds. It only remains to show that α' holds.

By (2), there exists some successor state of the current-state named by V_S such that one of the transitions $\overset{S}{\rho}_{\pi_1}$ is true together with α'. But because the transition system is deterministic, the next-state variables ${V_S}'$ are uniquely determined by the present-state variables V_S. In other words, the existence of a successor state which satisfies α' implies that every successor state satisfies α'. Since $\overset{S}{\rho}_{\pi_2}$ names a specific successor state, this successor state satisfies α'. □

3.3 CVC

We have integrated the Cooperating Validity Checker (CVC) system developed at Stanford University [SBD02] to check the verification conditions. CVC is an automatic theorem prover for 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. 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 generated by VOC-64. 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.

3.4 Example of VOC-64

The intermediate language of SGI Pro-64 (or SGI for short) is WHIRL. After each round of optimization, the compiler outputs ASCII formatted WHIRL code, which can be read by a parser and translated back into a graphic representation. A description of the tool is in [ZPFG03].

In Fig. 11 we present a program and several of the optimizations applied to it.

In the verification conditions that VOC-64 produced for each stage of the optimization are presented below. We use annotation of (a)–(e) to denote the state of the optimizations the variables/conditions refer to. So, for example, C^{cd}_{01} refers to the verification condition of the target program (d) going from `B0` to `B1` relative to the source program (c), and y_e denotes the y variable of program (e). The variable π is the control variable denoting the program counter of the program. The data mapping and assertion are given by:

$$\begin{array}{l}
\alpha_{ab} : (\pi_b \in \{1,2\} \ \rightarrow \ (n_a = n_b) \wedge (w_a = w_b) \wedge (y_a = y_b)) \\
\alpha_{bc} : (\pi_c \in \{1,2\} \ \rightarrow \ (n_b = n_c) \wedge (w_b = w_c) \wedge (y_b = y_c)) \\
\alpha_{cd} : (\pi_d \in \{1,2\} \ \rightarrow \ (n_c = 500) \wedge (w_c = w_d) \wedge (y_c = y_d)) \\
\alpha_{de} : (\pi_d \in \{1,2\} \ \rightarrow \ (w_d = w_e) \wedge (y_d = y_e)) \\
\varphi_3 : \ (.t1 = 2 \cdot y_e)
\end{array}$$

The verification considtions generated by VOC-64 for the example are listec in Fig. 12.

```
B0
   n <- 500
   y <- 0
   w <- 1
B1
   WHILE (n >= w)
B2
   w <- w + y * 2 + 3
   y <- y + 1
B3
   RETURN
```

(a) Input Program

```
B0
    n <- 500
    y <- 0
    w <- 1
    IF !(n >= w) GOTO L2
B1
L1: w <- w + y * 2 + 3
    y <- y + 1
    IF (n >= w) GOTO L1
B2
L2: RETURN
```

(b) After Loop Inversion

```
B0
    n <- 500
    y <- 0
    w <- 1
B1
L3: w <- w + y * 2 + 3
    y <- y + 1
    IF (n >= w) GOTO L3
B2
    RETURN
```

(c) After Constant Folding

```
B0
    y <- 0
    w <- 1
B1
L3: w <- w + y * 2 + 3
    y <- y + 1
    IF (w <= 500) GOTO L3
B2
    RETURN
```

(d) After Copy Propagation and Dead Code Elimination

```
B0
    y <- 0
    w <- 1
    .t1 <- 0
B1
L3: w <- .t1 + w + 3
    y <- y + 1
    .t1 <- .t1 + 2
    IF (w <= 500) GOTO L3
B2
    RETURN
```

(e) After Strength Reduction

Fig. 11. Stages of Optimization

$$
\begin{array}{ll}
C^{ab}_{01}: & (\pi_b = 0 \land \pi'_b = 1 \land n'_b = 500 \land w'_b = 1 \land y'_b = 0 \land 500 \geq 1 \land \alpha_{ab}) \rightarrow \\
& (\pi_a = 0 \land \pi'_a = 2 \land n'_a = 500 \land w'_a = 1 \land y'_a = 0 \land 500 \geq 1 \land \alpha'_{ab}) \\
C^{ab}_{02}: & (\pi_b = 0 \land \pi'_b = 2 \land n'_b = 500 \land w'_b = 1 \land y'_b = 0 \land \neg(500 \geq 1) \land \alpha_{ab}) \rightarrow \\
& (\pi_a = 0 \land \pi'_a = 3 \land n'_a = 500 \land w'_a = 1 \land y'_a = 0 \land \neg(500 \geq 1) \land \alpha'_{ab}) \\
C^{ab}_{11}: & (\pi_b = 1 \land \pi'_b = 1 \land w'_b = w_b + 2 \cdot y_b + 3 \land y'_b = y_b + 1 \land (n_b \geq w_b + 2 \cdot y_b + 3) \land \alpha_{ab}) \rightarrow \\
& (\pi_a = 2 \land \pi'_a = 2 \land w'_a = w_a + 2 \cdot y_a + 3 \land y'_a = y_a + 1 \land (n_a \geq w_a + 2 \cdot y_a + 3) \land \alpha'_{ab}) \\
C^{ab}_{12}: & (\pi_b = 1 \land \pi'_b = 2 \land w'_b = w_b + 2 \cdot y_b + 3 \land y'_b = y_b + 1 \land \neg(n_b \geq w_b + 2 \cdot y_b + 3) \land \alpha_{ab}) \rightarrow \\
& (\pi_a = 2 \land \pi'_a = 3 \land w'_a = w_a + 2 \cdot y_a + 3 \land y'_a = y_a + 1 \land \neg(n_a \geq w_a + 2 \cdot y_a + 3) \land \alpha'_{ab}) \\
C^{bc}_{01}: & (\pi_c = 0 \land \pi'_c = 1 \land n'_c = 500 \land w'_c = 1 \land y'_c = 0 \land \alpha_{bc}) \rightarrow \\
& (\pi_b = 0 \land \pi'_b = 1 \land n'_b = 500 \land w'_b = 1 \land y'_b = 0 \land 500 \geq 1 \land \alpha'_{bc}) \\
C^{bc}_{11}: & (\pi_c = 1 \land \pi'_c = 1 \land w'_c = w_c + 2 \cdot y_c + 3 \land y'_c = y_c + 1 \land (n_c \geq w_c + 2 \cdot y_c + 3) \land \alpha_{bc}) \rightarrow \\
& (\pi_b = 1 \land \pi'_b = 1 \land w'_b = w_b + 2 \cdot y_b + 3 \land y'_b = y_b + 1 \land (n_b \geq w_b + 2 \cdot y_b + 3) \land \alpha'_{bc}) \\
C^{bc}_{12}: & (\pi_c = 1 \land \pi'_c = 2 \land w'_c = w_c + 2 \cdot y_c + 3 \land y'_c = y_c + 1 \land \neg(n_b \geq w_c + 2 \cdot y_c + 3) \land \alpha_{bc}) \rightarrow \\
& (\pi_b = 1 \land \pi'_b = 2 \land w'_b = w_b + 2 \cdot y_b + 3 \land y'_b = y_b + 1 \land \neg(n_b \geq w_b + 2 \cdot y_b + 3) \land \alpha'_{bc}) \\
C^{cd}_{01}: & (\pi_d = 0 \land \pi'_d = 1 \land w'_d = 1 \land y'_d = 0 \land \alpha_{cd}) \rightarrow \\
& (\pi_c = 0 \land \pi'_c = 1 \land n'_c = 500 \land w'_c = 1 \land y'_c = 0 \land \alpha'_{cd}) \\
C^{cd}_{11}: & (\pi_d = 1 \land \pi'_d = 1 \land w'_d = w_d + 2 \cdot y_d + 3 \land y'_d = y_d + 1 \land (500 \geq w_d + 2 \cdot y_d + 3) \land \alpha_{cd}) \rightarrow \\
& (\pi_c = 1 \land \pi'_c = 1 \land w'_c = w_c + 2 \cdot y_c + 3 \land y'_c = y_c + 1 \land (n_c \geq w_c + 2 \cdot y_c + 3) \land \alpha'_{cd}) \\
C^{cd}_{12}: & (\pi_d = 1 \land \pi'_d = 2 \land w'_d = w_d + 2 \cdot y_d + 3 \land y'_d = y_d + 1 \land \neg(500 \geq w_d + 2 \cdot y_d + 3) \land \alpha_{cd}) \rightarrow \\
& (\pi_c = 1 \land \pi'_c = 2 \land w'_c = w_c + 2 \cdot y_c + 3 \land y'_c = y_c + 1 \land \neg(n_c \geq w_c + 2 \cdot y_c + 3) \land \alpha'_{cd}) \\
C^{de}_{01}: & (\pi_e = 0 \land \pi'_e = 1 \land .t1' = 0 \land w'_e = 1 \land y'_e = 0 \land \alpha_{de}) \rightarrow \\
& (\pi_d = 0 \land \pi'_d = 1 \land w'_d = 1 \land y'_d = 0 \land \alpha'_{de} \land \varphi') \\
C^{de}_{11}: & (\pi_e = 1 \land \pi'_e = 1 \land .t1' = .t1 + 2 \land w'_e = .t1 + w_e + 3 \land y'_e = y_e + 1 \land (500 \geq .t1 + w + 3) \land \alpha_{de} \land \varphi) \rightarrow \\
& (\pi_d = 1 \land \pi'_d = 1 \land w'_d = w_d + 2 \cdot y_d + 3 \land y'_d = y_d + 1 \land (500 \geq w_d + 2 \cdot y_d + 3) \land \alpha'_{de} \land \varphi') \\
C^{de}_{12}: & (\pi_e = 1 \land \pi'_e = 2 \land .t1' = .t1 + 2 \land w'_e = .t1 + w_e + 3 \land y'_e = y_e + 1 \land \neg(500 \geq .t1 + w + 3) \land \alpha_{de} \land \varphi) \rightarrow \\
& (\pi_d = 1 \land \pi'_d = 2 \land w'_d = w_d + 2 \cdot y_d + 3 \land y'_d = y_d + 1 \land \neg(500 \geq w_d + 2 \cdot y_d + 3) \land \alpha'_{de})
\end{array}
$$

Fig. 12. Verification Conditions Generated by VOC-64

4 Run-time Validation of Speculative Optimizations

This section gives 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.

Unlike compiler validation, as discussed in previous sections, 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 will be 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 will be necessary to jump to an unoptimized version of the code.

The particular optimizations to be addressed here are the ones discussed in Section 2. As shown in [GHCP02], software pipelining has also been shown to be particularly amenable to run-time validation.

The work presented here is somewhat preliminary. Thus, we proceed primarily by example, without presenting an entire formalism for validation of speculative optimizations.

4.1 Formal Basis

Suppose we have a loop reordering that performs the transformation

for I = 1, N do		for i = $P^{-1}(1)$ ... $P^{-1}(N)$ do
a[D(I)] = ...	$\Longrightarrow$	a[D(i)] = ...
... = ... a[U(I)] ...		... = ... a[U(i)] ...

where P is a permutation determining the sequence of values taken on by the index variable i the transformed loop. That is, $P(i) = j$ iff the index variable takes on the value i in the jth iteration of the transformed loop. Thus, $P^{-1}(j)$ gives the value of the index variable in the jth iteration of the transformed loop.

For run-time validation of this transformation we will use, as an invariant to be maintained at run-time, a version of the simplified rule presented in the previous section as follows:

$$\forall i, j \leq N : \; i < j \;\wedge\; D(i) = U(j) \Longrightarrow P(i) < P(j) \tag{4}$$

Intuitively, this says that if `a[`$D(i)$`]` and `a[`$U(j)$`]` refer to the same location, such that the write to that location would have occurred before the read from that location in the original loop, the write must also occur before the read in the transformed loop.

This rule accounts for perfectly nested loops as well, of the form,

```
for i_1,N_1 do
 ...
  for i_m = 1,N_m do
     a[D(i_1,...,i_m)] = ...
      ...  = ... a[U(i_1,...,i_m)] ...
```

where D and U each return a vector giving the indices into multi-dimensional array `a`.

Rule (4), above, actually governs only the preservation of flow dependence (*aka* true dependence). The complete rule, accounting for anti- and output dependence as well is:

$$\forall i, j \leq N : \; i < j \;\wedge\; (D(i) = U(j) \;\vee\; U(i) = D(j) \;\vee\; D(i) = D(j)) \Longrightarrow P(i) < P(j)$$

For simplicity of the presentation, we'll only consider the simpler rule here.

4.2 Safety Properties of Run-Time Validation

The safety properties that must be preserved by the run-time test are:

1. The test must be able to determine, either precisely or conservatively, if a dependence may be violated by the optimized loop.
2. Once a run-time test determines that a dependence may be violated by the optimized loop, there must be an execution path that can be taken to produce the correct result.
3. The run-time test must be able to determine that a dependence may be violated *before* the dependence has actually been violated.

This last property, which we refer to as the *testability property*, is, perhaps, overly strict. One can imagine detecting a dependence violation after the violation has occurred, and executing patch-up code to undo the effect of the violation. We do not take this approach, however.

4.3 Efficiency Issues for Run-Time Validation

In order to be worthwhile, run-time validation should satisfy the following qualitative properties:

1. The run-time test should occur as infrequently as possible.
2. The test should be as inexpensive as possible, in terms of time and space.
3. The cost of executing the loop when a potential dependence violation is detected should be no greater than the cost of the orginal (unoptimized) loop.

We've developed a set of run-time validation techniques which satisfy these properties to various extents. In this paper, though, we concentrate on describing how run-time tests are used to preserve dependences, rather than addressing their efficiency.

4.4 The Testability Property

The testability property, which states that a potential dependence violation must be detected before the violation actually occurs, is what makes run-time validation particularly difficult in many cases. Without this constraint, there is a reasonably efficient algorithm for testing to see if a transformed loop satisfies rule (4), above. This algorithm is:

Input: Permutation P, functions D, U with ranges $\{1..m\}$
Output: "success" or "failure"
Data structure: `MARK: array[1..m] of integer`, with all elements initialized to zero.
Algorithm:

```
for k := P^{-1}(1), ..., P^{-1}(N) do
   MARK[U( k)] := max(Mark[U( k)], k)
   if MARK[D(k)] ≥ k then exit with "failure"
end
exit with "success"
```

To see why this algorithm detects a violation of rule (4), note that such a violation can only occur when there exist i and j such that $(i < j) \wedge (D(i) = U(j)) \wedge (P(i) \geq P(j))$. This is exactly the situation detected by the algorithm, because:

1. If `k` takes on a value i such that `MARK[`$D(i)$`]` contains a non-zero value j, then it must be the case that $D(i) = U(j)$ because they produce the same index into the `MARK` array. Further, since j was written into `MARK[`$U(j)$`]` before being read as `MARK[`$D(i)$`]` by the algorithm, `k` must have taken on the value j before it took on the value i. Therefore, $P(j) < P(i)$.
2. If $j > i$, the rule is violated and the algorithm exits with failure.

The use of `max` in the algorithm is necessary if U(`k`) could produce the same value for several different values of `k`. This algorithm has several desirable properties, namely:

1. The loop index variable `k` iterates over the sequence $P^{-1}(1), \ldots, P^{-1}(N)$, just as it does in the transformed loop, above.
2. The computation of D(`k`) and U(`k`) is the same computation performed by the transformed loop.

Together, these properties mean that a run-time algorithm satisifying them can be integrated into the transformed loop code without a) changing the order of the transformed loop or adding an additional loop and b) having to compute $D(i)$ or $U(i)$ for the sole purpose of the run-time test. In particular, the transformed loop with the run-time test might look like:

```
for i = P^{-1}(1), ... , P^{-1}(N) do
    w = D(i)
    r = U(i)
    MARK[r] = max(MARK[r],i)
    if MARK[w] >= i then goto patchup_code
    a[w] = ...
    ...  = ... a[r] ...
end
```

However, the algorithm has two undesirable properties, namely:

1. It requires an array `MARK` of a size equal to the size of `a`, as well as $2N$ accesses to `MARK`.
2. Worst of all, in the cases where the rule is violated, the algorithm doesn't detect the violation until the write is about to happen – which isn't until after the read has already occurred on a previous iteration.

This last characteristic renders the algorithm useless as a tool for run-time validation, because it does not exhibit the testability property. Intuitively, the problem is that the incorrect value read in a previous iteration may have been used in subsequent iterations, thus rendering the program incorrect.

Does there exist an algorithm that has the desirable properties we want without the undesirable properties listed above? If, as in the code above, we require that the run-time test be integrated into the optimized loop and that each $D(j)$ be computed only during the iteration in which the loop index has the value j, then the answer is "no". To see this, it is sufficient to note that to satisfy the testability property, we would have to be able, in each ith iteration, to answer the question

"Is there a value $j \in \{P^{-1}(i+1), \ldots, P^{-1}(N)\}$ such that $j < P^{-1}(i)$ and $D(j) = U(P^{-1}(i))$?"

without computing $D(j)$. For arbitrary function D and permutation P, this is not possible.

4.5 Restricting *D*

So far, we've placed no restrictions on the functions D and U. That is, we allow D and U to be sufficiently complex as to prevent static analysis. However in most loops it is not the case that both D and U cannot be analyzed. For example, the transformation of a loop of the form

```
for i = 1, N do                    ⟹    for i = P^{-1}(1), P^{-1}(N) do
    a[i] = ...                               a[i] = ...
    ...  = ... a[U(i)] ...                   ...  = ... a[U(i)] ...
```

where $U(i)$ is arbitrarily complex, is not a candidate for static analysis but does satisfy the testability property[5]. A version of the transformed code with the appropriate run-time test might be:

```
for i = P^{-1}(1), P^{-1}(N) do
    r = U(i)
    if test(i,r) goto escape_code
    a[i] = ...
    ...  = ... a[r] ...
end
```

where

$\texttt{test}(i,r) \;=\; r < i \;\wedge\; P(r) > P(i).$

Although this test function appears prohibitively expensive, in practice it is not. The compiler, having created the permutation P, can generate efficient code for the test that is specific for P. For example, if the transformation is loop reversal then the test function is simply $\texttt{test}(i,r) = (r < i)$. If the transformation is loop interchange for two nested loops, then the test function is:

$$\begin{aligned}\texttt{test}((i_1,i_2),(r_1,r_2)) &= (r_1,r_2) \prec_{lex} (i_1,i_2) \;\wedge\; (i_2,i_1) \preceq_{lex} (r_2,r_1) \\ &= r_1 < i_1 \;\wedge\; r_2 > i_2.\end{aligned}$$

4.6 Examples

Dynamic Loop Interchange In this section, we show how the run-time test for a particular loop interchange example can be automatically derived from the formulation of the general interchange test given above. Suppose the tranformation is:

```
for i = 1, N do                        for j = 1, M do
    for j = 1, M do                        k = 10 - j
        k = 10 - j             ⟹           for i = 1, N do
        a[i, j] = a[i-1, j-k] + c              a[i, j] = a[i-1, j-k] + c
```

[5] Note that, in this case, the testability property only holds if we restrict our concern to flow dependence and ignore anti-dependence. A constraint that would allow us to ignore anti-dependence, for example, is $U(i) < i$.

The benefits to this loop interchange are 1) the computation of k was able to 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. Using the formulation of the general validation test in the for loop interchange, a test is inserted as follows:

```
for j = 1, M do
    k = 10 - j
    for i = 1, N do
        if test((i, j),(i-1,j-k)) goto escape_code
        a[i, j] = a[i-1, j-k] + c
```

where it is easy to see that $\mathtt{test}((i,j),(i-1,j-k)) = (i-1 < i) \wedge (j-k > j) = (k < 0)$. Thus, after inserting the actual test and moving it to the outer loop (since the test is invariant in the inner loop), we arrive at:

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

If the dependence is about to be violated because `k < 0`, then the program can transfer control to a copy of the orginal loop to execute the remainder of the iterations. The remaining iterations of the original loop can be executed by:

```
escape_code:

for ii = 1, N do
    for jj = j, M do
        k = 10 - jj
        a[ii, jj] = a[ii-1, jj-k] + c
```

where j is the same variable as in the transformed loop, above, containing the last value that j took on within the optimized loop.

Dynamic Tiling In the previous example, we were able to generate a version of the original loop that could be executed once the test in the transformed loop detected an impending dependence violation. For a more complicated loop optimization, such as tiling, it may be easier to simply adjust the transformed loop based on a run-time test prior to entering the transformed loop. For example, consider the loop tiling transformation in Fig. 13.

```
for i = 1, N do
    for j = 1, N do
    a[i, j] = a[i-k, j+10] + c
```

$\Longrightarrow$

```
for ii = 1, N by b do
    for jj = 1, N by b do
        for i = ii to ii+b-1
            for j = jj, jj+b-1 do
                a[i, j] = a[i-k, j+10] + c
```

Fig. 13. Tiling Transformation

where k is a variable whose value is unknown at compile-time. Without a run-time test, this transformation is not necessarily correct. However, it is correct under the condition that k $\geq$ b. Thus, if a fixed tile size b is required, the compiler can generate code to test the value of k prior to entering the tiled loop and to jump to the original loop if $k < b$. Another alternative is to simply add the assignment b = min(k,$maxTileSize$) before the tiled loop.

4.7 Architectural Considerations

Adding run-time tests to validate compiler optimizations is becoming increasingly tractable due to, among other things, the emergence of new classes of processors that exhibit 1) the ability to exploit instruction-level parallelism (ILP) and 2) hardware features that reduce the cost of run-time tests and the compensation code when a dependence is about to be violated.

Most modern processors have multiple functional units for exploiting instruction level parallelism, either via dynamic scheduling of multiple instructions simultaneously on superscalar machines or via compiler-specified multi-operation instructions on VLIW/EPIC processors. In fact, the challenge with these machines has been to find sufficient ILP in programs in order to fully utilize all functional units each cycle. Thus, the additional tests required for run-time validation can often be scheduled in unused slots in the instruction schedule.

Some hardware features that aid in testing and compensating for dependence violations can be found on the VLIW/EPIC class of machines exemplified by the Intel IA-64 architecture. The *dynamic disambiguation* feature, which provides several instructions that test for aliasing, can be used to implement low-cost run-time validation tests. Once a run-time test has determined that a dependence violation is about to occur, the *predication* feature of the IA-64 can be used to disable instructions in such as way as to preserve the correctness of the code. For more detail on this last point, as applied to run-time validation of software pipelining, see [GHCP02].

5 Conclusion

We reviewed the translation validation approach, its advantages, the main proof rule used for structure preserving transformations, and described a tool, VOC-64, that generates verification conditions for SGI Pro-64.

We then turn to optimizations that cause major changes in the structure of the code. For reordering transformations we propose special permutation loops that can easily deal with the most common optimizations. For transformations whose validity is hard, or even impossible, to check at compile-time, we propose a *run-time translation validation* that allows for aggressive optimizations, while guaranteeing that no dangerous aliasing occurs. When a problem is detected, the code escapes to an unoptimized version of the original loop, where it completes the computation in a slower, but guaranteed correct, manner. This approach seems promising, and we are currently working on using CVC to automate the process of generating such speculative optimizations.

References

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

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

Fre02. C.C. Frederiksen. Correctness of Classical Compiler Optimizations using CTL. In *Proc. of Compiler Optimization meets Compiler Verificaiton (COCV) 2002*, Electronic Notes in Theoretical Computer Science (ENTCS), volume 65, issue 2.

FORS01. : No match. J.C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Proc.* 13^{rd} *Intl. Conference on Computer Aided Verification (CAV'01), volume 2102 of*Lect. Notes in Comp. Sci., *Springer-Verlag*, 2001.

GGB02. S. Glesner, R. Geiß and B. Boesler. Verified Code Generation for Embedded Systems. In *Proc. of Compiler Optimization meets Compiler Verificaiton (COCV) 2002*, Electronic Notes in Theoretical Computer Science (ENTCS), volume 65, issue 2.

GHCP02. B. Goldberg, C. Huneycutt, E. Chapman, and K. Palem. Software bubbles: Using predication to compensate for aliasing in software pipelines. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002.

MAB+94. Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

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.

PRSS99. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In *CAV'99*, pages 455–469, 1999.

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.

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.

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

PZP00. A. Pnueli, L. Zuck, and P. Pandya. Translation validation of optimizing compilers by computational induction. Technical report, Courant Institute of Mathematical Sciences, New York University, 2000.

Riv03. X. Rival. Abstract Interpretation-Based Certification of Assembly Code. In *Proc. 4^{th} Intl. Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pages 41–55, 2003.

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.

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.

SOR93. N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.

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.

ZPL00. L. Zuck, A. Pnueli, and R. Leviathan. Validations of optimizing compliers. Technical report, Weizmann Institute of Science, 2000.