Theory and Algorithms for the Generation and Validation of Speculative Loop Optimizations

Ying Hu Clark Barrett Benjamin Goldberg
Department of Computer Science
New York University
{yinghu,barrett,goldberg}@cs.nyu.edu

Abstract

Translation validation is a technique that verifies the results of every run of a translator, such as a compiler, instead of the translator itself. Previous papers by the authors and others have described translation validation for compilers that perform loop optimizations (such as interchange, tiling, fusion, etc), using a proof rule that treats loop optimizations as permutations.

In this paper, we describe an improved permutation proof rule which considers the initial conditions and invariant conditions of the loop. This new proof rule not only improves the validation process for compile-time optimizations, it can also be used to ensure the correctness of speculative loop optimizations, the aggressive optimizations which are only correct under certain conditions that cannot be known at compile time. Based on the new permutation rule, with the help of an automatic theorem prover, CVC Lite, an algorithm is proposed for validating loop optimizations. The same permutation proof rule can also be used (within a compiler, for example) to generate the runtime tests necessary to support speculative optimizations.

Key words: Compiler validation, speculative loop optimizations, translation validation, formal methods.

1. Introduction

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

There has been considerable work [11, 9, 14, 15, 8] in this area to develop TV techniques for optimizing compilers that utilize *structure preserving* transformations, i.e. transformations which do not greatly change the structure of the program (e.g. dead code elimination, loop-invariant code motion, copy propagation) [1, 12] as well as *structure modifying* transformations, such as reordering loop transformations (e.g. interchange, tiling), that do significantly change the structure of the program [2, 13].

For the translation validation of reordering loop transformations, in previous publications [15, 4, 6], the authors and others have proposed a proof rule PERMUTE that treats loop transformations as permutations. Although the PERMUTE rule has been used to check the validity of a number of reordering loop transformations, it has the limitation of requiring the loop transformations to be valid in all contexts without considering any conditions outside of the loop. In this paper, we introduce an improved permutation proof rule Inv-Permute which considers the context of the loop and thus is more powerful than the PERMUTE rule.

The new permutation rule Inv-Permute can be used by a compiler to decide whether some loop transformation is valid at compile time given a loop invariant determined by static analysis. Because an appropriate invariant is generally hard to find, we use an automatic theorem prover, CVC Lite [3], to try to generate a condition under which the loop transformation is valid. This condition can then be checked in the loop to see whether it is indeed invariant. This paper gives an algorithm for generating such a condition using CVC Lite.

In some cases, it is impossible to determine at compilation time whether a desired loop optimization is legal. This is usually because of limited capability to check effectively that syntactically different array index expressions refer to the same array location. In such cases, the validation condition derived by CVC Lite cannot be proved to hold at compile-time, but it may hold at run time. One possible remedy to this situation is to perform the optimization conditionally, by adding code to check at run time whether

the loop optimization is safe. If the run-time check fails, the code chooses to use an unoptimized version of the loop which completes the computation in a manner which may be slower but is guaranteed to be correct. An algorithm for generating the run-time test for speculative loop optimizations was given in a previous paper [4]. This paper extends the work described in [4] in the following ways:

- The proof rule Inv-Permute is introduced as the formal basis for using loop invariants to validate loop optimizations (both speculative and non-speculative).
- The algorithm for finding invariants needed to apply rule Inv-Permute has been improved and is described in more detail.
- More examples and results are given.

This paper is organized as follows. Section 2 describes the new proof rule Inv-Permute for reordering loop transformations. Section 3 describes an improved algorithm for determining a sufficient condition under which an otherwise invalid transformation may be applied. Using the proof rule, we show that such a transformation is valid if the condition can be statically verified. Alternatively, a run-time test for the condition can be inserted. Section 4 gives several examples and shows the results of applying the algorithms in Section 3 to these examples. Finally, Section 5 concludes.

2. The proof rule

A modern compiler performs a set of advanced optimizations to make the compiled code run faster. Among them are loop optimizations which improve parallelism and make efficient use of the memory hierarchy. A *reordering transformation* is defined [2] as any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statement. Many loop transformations, including reversal, fusion, distribution, interchange, and tiling, are in the class of reordering transformations.

Traditionally, dependence analysis has been used to determine whether it is safe to perform certain kinds of program transformations. In the presence of two accesses to the same memory location (where at least one is a write) dependence theory [2] states that a reordering transformation *preserves* a dependence if it preserves the relative execution order of the source and target (i.e. the first memory access and second memory access) of that dependence. A reordering transformation is *valid* if it preserves all dependences in the program. To decide whether a reordering loop transformation preserves the meaning of the program, the compiler usually performs dependence analysis. The basic idea is that for any pair of statements s_1 and s_2, if there is any dependence between them, then the order of executing them cannot be changed in the transformation. Let $Dependence(s_1, s_2)$ be a predicate denoting whether there is any dependence between the statements s_1 and s_2, and let $Reorder(s_1, s_2)$ be a predicate denoting whether the transformation can safely reorder the execution of s_1 and s_2. The dependence rule can be schematized as:

$$Dependence(s_1, s_2) \implies \neg Reorder(s_1, s_2) \quad (1)$$

or, equivalently,

$$Reorder(s_1, s_2) \implies \neg Dependence(s_1, s_2) \quad (2)$$

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

$$Reorder(s_1, s_2) \implies s_1; s_2; \sim s_2; s_1; \quad (3)$$

Rule (3) is more powerful than rule (2), because it validates more cases than rule (2). To formalize the idea, the rule PERMUTE [15] was proposed, which we review in this paper, and then extend.

Consider the generic loop in Fig. 1.

---

for $i_1 = L_1, H_1$ **do**
$\cdots$
for $i_m = L_m, H_m$ **do**
$B(i_1, \ldots, i_m)$

Figure 1: A General Loop

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

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

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 $\mathcal{I}$.

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

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

Figure 2: Rule PERMUTE for reordering loop transformations

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

Figure 3: A loop interchange example

Consider a generic loop transformation:

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

$$
\Longrightarrow
$$

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

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

The PERMUTE rule in Fig. 2 has two requirements to validate the reordering loop transformation:

1. The mapping F is a bijection from $\mathcal{J}$ onto $\mathcal{I}$.
2. For every pair of loop index vectors $\vec{i}_1, \vec{i}_2$, such that the order of executing $\mathtt{B}(\vec{i}_1), \mathtt{B}(\vec{i}_2)$ is reversed after the transformation, the result of executing the pair of iterations in either the original or the reversed order is the same.

The symbol $\sim$ in Fig. 2 means that two pieces of code are equivalent, i.e. they transform the program state in the same way. In addition to being more general (as described above), the PERMUTE rule has two additional advantages over the standard dependence analysis approach. First, it only needs the information inside the loop to generate the logical formula for code equivalence, without explicitly having to perform dependence analysis. Second, PERMUTE can leave the task of proving the legality of transformations to an automatic theorem prover, which can not only determine whether a transformation is legal, but can actually provide a proof in the case that it is[1].

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

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

$$
\Longrightarrow
$$

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

where the representation $\{\phi\}$ uses Hoare's precondition notation [7], meaning that we assume ϕ holds before each of the two pieces of code.

It is important that the invariant ϕ hold at the beginning of the loop and continue to hold (i.e. be invariant) during the execution of the loop. We also require that ϕ does not contain any loop index variables, otherwise it may become invalid by the updating of loop index variables at the end of each iteration. Fig. 4 gives the improved Inv-Permute rule, which includes an invariant ϕ assumed to not contain any reference to the loop index variables.

In Inv-Permute, premises 1 and 2 ensure that the permutation F is a bijection, premise 3 ensures that the property ϕ holds at the beginning and end of each iteration of the loop,

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

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

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

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

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

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

Figure 4: Rule Inv-Permute for reordering loop transformations

and premise 4 ensures the equivalence of the source and target loop by commutativity. The PERMUTE rule can be regarded as a weaker version of the Inv-Permute rule with invariant $\phi = true$.

The following lemma directly implies the soundness of the Inv-Permute rule:

Lemma 2.1 (Soundess of Inv-Permute) *Let $\mathcal{I}$ and $\mathcal{J}$ be finite sets ordered by $\prec_{\mathcal{I}}$ and $\prec_{\mathcal{J}}$ respectively such that $|\mathcal{I}| = |\mathcal{J}|$. Let $F\colon \mathcal{J} \mapsto \mathcal{I}$ be a bijection. Let ϕ be a property independent of the loop index variables. If*

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

and

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

then

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

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

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

then

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

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

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

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

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

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

Notice that the first assumption

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

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

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

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

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

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

Example Let ϕ be the property $k \geq 0$. For the example in Fig. 3, let the the loop index vector $\vec{i_1}$ be the tuple (i_1, j_1), and $\vec{i_2}$ the tuple (i_2, j_2). The domain $\mathcal{I}$ is $[1, N] \times [1, M]$, the domain $\mathcal{J}$ is $[1, M] \times [1, N]$, the permutation function F is $F((j, i)) = (i, j)$, and the body $B((i, j))$ is $A[i + k, j + 1] = A[i, j] + 1;$. The Inv-Permute rule requires:

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

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

$$\Longrightarrow$$

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

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

$$\sim$$

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

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

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

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

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

$$\Longrightarrow$$

$$k \geq 0 \Longrightarrow$$

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

$$\Longrightarrow$$

$$A_2 = A'_2$$

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

3. Using the proof rule

This section gives the algorithms for using the proof rule Inv-Permute to validate loop transformations and to generate run-time tests for speculative loop optimizations.

3.1. Loop transformations with invariants

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

1. Apply Inv-Permute for the loop under $\phi = true$, and generate the verification condition θ for premise 4.
2. Check the validity of θ using CVC Lite. If it is valid, exit with a positive result.
3. Otherwise, from the counter-example ψ produced by CVC Lite, attempt to infer a condition ϕ that makes the verification condition valid. If no appropriate ϕ is found exit with a negative result.
4. Analyze the context statically to check whether ϕ holds as the initial condition and is loop invariant. If ϕ holds, exit with a positive result.
5. Otherwise, exit with a negative result.

Step 3 will be explained in more detail in Section 3.3.

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

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

Figure 5: An example for speculative loop interchange

3.2. Speculative loop optimizations

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

To preserve the benefit of loop optimizations in the presence of variables whose values cannot be determined statically, a run-time test, testing the values of various variables used in the loop, can be inserted into the compiled program. Loop optimizations enabled in this manner are called *speculative* loop optimizations[2]. The idea of validating speculative loop optimizations in the TV framework was introduced in [4]. However since the Inv-Permute rule was not established in that paper, only the concept and some heuristics were given. In this paper, Section 2 described the new proof rule Inv-Permute, which is the formal basis for validating speculative loop optimizations. Fig. 5 shows the result of applying a speculative loop optimization to the interchange example of Fig. 3.

With the Inv-Permute rule, the scheme for speculative loop optimizations is:

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

3.3. Automatically generating invariants using CVC Lite

We implement Step 3 in the previous algorithms as follows:

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

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

The following are some observations: As the invariant ϕ must refer to non-index variables, at least one such variable must be in the chosen formula. Because equality is usually a stricter requirement than disequality, the chosen formula should not be a disequality such as $\neg(x = y)$. Also a formula including array elements may not be a good choice (as they generally include index variables that may be hard to eliminate).

For the validity of interchanging the loop example in Fig. 3, CVC Lite generates the following counter example with six formulas:

2 This technique was called *inspector/executor* in [5, 10].

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

Figure 6: A fusion example

1.) $0 < i_2 - i_1$
2.) $(i_2 + k, 1 + j_2) = (i_1, j_1)$
3.) NOT $(i_2 + k, 1 + j_1) = (i_2, j_2)$
4.) NOT $(-k = 0)$
5.) NOT $(i_2 + k, 1 + j_1) = (i_2 + k, 1 + j_2)$
6.) NOT $(i_2, j_1) = (i_2 + k, 1 + j_2)$

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

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

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

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

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

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

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

$\alpha:\ i_1 - i_2 < 0\ \wedge\ i_1 - i_2 > -N$
$\wedge\ j_1 - j_2 > 0\ \wedge\ j_1 - j_2 < N$
and
$\beta:\ A_1 = write(A, (i_1 + k, j_1 + 1), A[i_1, j_1] + 1)$
$\wedge\ A_2 = write(A_1, (i_2 + k, j_2 + 1), A_1[i_2, j_2] + 1)$
$\wedge\ A'_1 = write(A, (i_2 + k, j_2 + 1), A[i_2, j_2] + 1)$
$\wedge\ A'_2 = write(A'_1, (i_1 + k, j_1 + 1), A'_1[i_1, j_1] + 1)$
$\longrightarrow \quad A_2 = A'_2,$

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

4. Results

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

The first example is the fusion example in Fig. 6, where two simple loops in the source are merged into one simple loop in the target. Observe that this transformation is legal when no anti-dependence is created during the merging of the loops, i.e it is legal when $(k \geq 0)$ or $(|k| \geq N)$. Notice that the two loops in the source can be treated as a general loop structure [6], such that rule Inv-Permute can be applied. The following table gives the logical formulas for α, β, ϕ, and ϕ'. The result shows that the fusion is valid when $k \geq 0 \vee k \leq -N$, which is consistent with the above observation.

$\alpha:$	$i_1 - i_2 > 0\ \wedge\ i_1 - i_2 < N$
$\beta:$	$A_1 = write(A, i_1, i_1)\ \wedge\ y = A_1[i_2 - k] \wedge$ $y' = A[i_2 - k]\ \wedge\ A'_1 = write(A, i_1, i_1)$ $\longrightarrow$ $A_1 = A'_1\ \wedge\ y = y'$
$\phi:$	$k - i_2 + i_1 \neq 0$
$\phi':$	$k \geq 0\ \vee\ k \leq -N.$

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

Figure 7: An interchange example

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

Figure 8: A reversal example

The second example is an interchange example given in Fig. 7, where the inner and outer loops in the source are exchanged in the target. Observe that this transformation is legal when the leftmost non-zero value in the dependence vector tuple (p, q) has the same sign ($pq \geq 0$) before and after interchange, or there is no loop-carried dependence ($|p| \geq N \ \vee \ |q| \geq |M|$). The following table gives the logical formulas for α, β, ϕ, and ϕ'. The result shows that the interchange is valid when

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

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

which is consistent with our observation.

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

The third example is the reversal example in Fig. 8, where the iteration order of the loop is reversed in the target. Observe that this transformation is legal when there is no loop-carried dependence ($k = 0 \ \vee \ |k| \geq N$). The following table gives the logical formulas for α, β, ϕ, and ϕ'. The result shows that here reversal is valid only when

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

which is consistent with our observation.

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

The fourth example is a tiling example given in Fig. 9, where the two-level loop is decomposed into a four-level loop such that the iteration space in the source is traversed in tiles of size 10×10 in the target. Observe that this transformation is legal when interchanging is legal ($k \geq 0$) or there is no dependence between the elements in the tiles of the same row ($|k| \geq 10$). The following table gives the logical formulas for α, β, ϕ, and ϕ', where ti_1, ti_2 are the row tile numbers, tj_1, tj_2 are the column tile numbers, and ri_1, ri_2, rj_1, rj_2 are the coordinates within the tiles. The result shows that the tiling is valid when

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

which is consistent with our observation.

$$\begin{array}{ll}
\alpha: & ri_1 - ri_2 < 0 \ \wedge \ ri_1 - ri_2 > -10 \wedge \\
& rj_1 - rj_2 < 10 \ \wedge \ rj_1 - rj_2 > -10 \wedge \\
& ti_1 - ti_2 = 0 \ \wedge \ tj_1 - tj_2 > 0 \\
\beta: & i_1 = 10 * ti_1 + ri_1 \ \wedge \ i_2 = 10 * ti_2 + ri_2 \wedge \\
& j_1 = 10 * tj_1 + rj_1 \ \wedge \ j_2 = 10 * tj_2 + rj_2 \wedge \\
& A_1 = write(A, (i_1, j_1), A[i_1 - k, j_1 - 1] + 1) \wedge \\
& A_2 = write(A_1, (i_2, j_2), A_1[i_2 - k, j_2 - 1] + 1) \wedge \\
& A'_1 = write(A, (i_2, j_2), A[i_2 - k, j_2 - 1] + 1) \wedge \\
& A'_2 = write(A'_1, (i_1, j_1), A'_1[i_1 - k, j_1 - 1] + 1) \\
& \qquad \longrightarrow \\
& \qquad A_2 = A'_2 \\
\phi: & -ri_2 + ri_1 - k \neq 0 \\
\phi': & k \geq 0 \ \vee \ k \leq -10
\end{array}$$

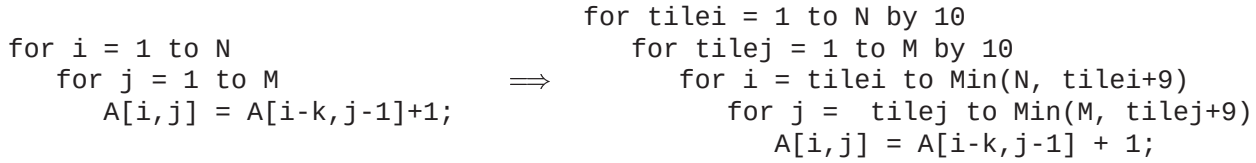

Figure 9: A tiling example

5. Conclusion

This paper began by reviewing the translation validation technique and the previous permutation proof rule for the translation validation of reordering loop optimizations. Then we introduced an improved permutation rule, which considers the initial and invariant conditions of the loop. This rule can be used by a compiler or a validator to check the correctness of the loop optimizations.

Based on the new permutation rule, with the help of an automatic theorem prover CVC Lite, an algorithm was proposed to generate the conditions which make the loop transformations valid. These conditions can be inserted as run-time tests for speculative optimizations. In this paper, we also showed the results of implementing the algorithm to generate the run-time tests for speculative loop optimizations by CVC Lite.

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

Acknowledgement. We would like to thank Amir Pnueli and Lenore Zuck for many helpful discussions and for their pioneering work in Translation Validation.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1988.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

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

[4] C. Barrett, B. Goldberg, and L. Zuck. Run-time validation of speculative optimizations using CVC. In O. Sokolsky and M. Viswanathan, editors, *Third International Workshop on Run-time Verification (RV)*, pages 87–105, July 2003. Boulder, Colorado, USA.

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

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

[7] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[8] Y. Hu, C. Barrett, B. Goldberg, and L. Zuck. TVOC: A tool for the translation validation of optimizing compilers. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, Apr. 2004.

[9] 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.

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

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

[12] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. of the SIGPLAN '91 Symp. on Programming Language Design and Implementation*, pages 33–44, 1991.

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

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

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