Into the Loops: Practical Issues in Translation Validation for Optimizing Compilers

Benjamin Goldberg   Lenore Zuck   Clark Barrett [1,2]

ACSys Group, Department of Computer Science, New York University

Abstract

Translation Validation is a technique for ensuring that the target code produced by a translator is a correct translation of the source code. Rather than verifying the translator itself, translation validation validates the correctness of each translation, generating a formal proof that it is indeed a correct. Recently, translation validation has been applied to prove the correctness of compilation in general, and optimizations in particular.

Tvoc, a tool for the Translation Validation of Optimizing Compilers developed by the authors and their collegues, successfully handles many optimizations employed by Intel's ORC compiler. Tvoc, however, is somewhat limited when dealing with loop reordering transformations. First, it relies on information, provided by the compiler, indicating which optimizations have been performed (in the case of the current ORC, this instrumentation is fortunately part of the compiler). Second, Tvoc has difficulties dealing with *combinations* of optimizations that are performed on the same block of code. A third limitation is that, in the theory upon which Tvoc is based, separate proof rules are needed for different categories of loop reordering transformations.

This paper addresses all the issues above. It presents a methdology for translation validation in the presence of combinations of optimizations, describes heuristics for determining which optimizations occurred (rather than relying on the compiler for this information), and presents a uniform proof rule that encompasses all reordering transformations performed by the Intel ORC compiler.

Key words: Translation validation, formal methods, loop optimizations, ORC, fusion, distribution.

[1] Email: {goldberg,zuck,barrett}@cs.nyu.edu

[2] This research was supported in part by and NSF grants CCR-0205571 and CCR-0098299 and and ONR grant N00014-99-1-0131.

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 (well-documented) difficulties of verifying an entire compiler, i.e. ensuring that it generates the correct target code for every possible valid source program, translation validation can be used to validate each run of the compiler, comparing the actual source and target codes.

There has been considerable work in this area, by these authors and others, to develop TV techniques for optimizing compilers that utilize *structure-preserving* transformations, i.e. optimizations which do not greatly change the structure of the program (e.g. dead code elimination, loop-invariant code motion, copy propagation) [ASU88,Muc97,WL91] as well as *structure-modifying* transformations, such as loop reordering optimizations (e.g. interchange, tiling), that do significantly change the structure of the program [AK02,Muc97,Wol95]. In previous publications, the authors and their students have described a prototype tool, Tvoc, that was developed for performing translation validation on the Intel Open Research Compiler (ORC) which performs a large number of transformations of both categories.

Although Tvoc is able to perform TV in the presence of a number of different structure-preserving and structure-modifying optimizations, it has suffered from the following drawbacks:

- TVOC has difficulty handling *combinations* of structure-preserving and structure-modifying optimizations. This is a serious drawback since often one transformation is performed on the code solely to enable a subsequent transformation.
- TVOC uses information produced by the compiler that indicates which loop optimizations have been performed. Fortunately, ORC does produce a file containing such information after every compilation, and no additional instrumentation of the compiler was required for TV. Although this information is never relied upon by TVOC to support a proof that the compilation is correct, it is used by TVOC to suggest the proof method to use on a particular section of code (because, as we describe subsequently in this paper, the TV of different classes of optimizations uses different proof rules).
- For validating loop reordering optimizations, TVOC has not used a single unifying proof rule, but rather has relied on several proof rules of different forms depending on the particular optimization that has been applied. Specifically, TVOC uses different proof rule for interchange, tiling, and skewing than it does for fusion and distribution. From a scientific (and engineering) perspective, a single proof-rule to handle all loop reordering optimizations would be more satisfying.

In this paper we describe our approaches to solving the above problems, which are currently being implemented. Briefly stated, the solutions are as follows:

- When presented with the target code T that reflects a series of transformations of the source code S, such that no intermediate versions of the code (e.g. after individual transformations) are available, TVOC will synthesize a series of intermediate versions of the code, based on what transformations it believes were performed. That is, it will generate synthetic intermediate versions $I_1, I_2, \ldots I_n$, which might possibly not have been created by the compiler at all. TVOC will then validate that the translation from S to I_1, the translation from I_j to I_{j+1} for each j, and the translation from I_n to T are correct.

- In order to avoid having TVOC rely on information produced by the compiler to determine which optimizations were actually performed, we have developed a set of heuristics that are used to generate this information given only the source and target codes. Heuristics were previously used in this way by Necula [Nec00] for the TV of structure-preserving transformations in GCC. In this paper, we describe the heuristics we use for the TV of structure-modifying transformations, specifically loop reordering optimizations.
- We have generalized the proof rule used for interchange, tiling, and skewing so that it now works for fusion and distribution as well. As a benefit of generalizing the proof rule, we are now able to use it in the validation of software pipelining.

The paper is structured as follows:

(i) Section 2 provides the necessary background for understanding our TV work in the validation of individual structure modifying transformations.

(ii) Section 3 describes how the proof rule that we have used for loop optimizations such as interchange and tiling can be used to validate loop fusion, loop distribution, and loop alignments, loop peeling, and loop unrolling.

(iii) Section 4 describes the kinds of combinations of optimizations that ORC performs, and our techniques for TV of such combinations using the creation of synthetic intermediate versions of the code.

(iv) Section 5 presents the heuristics that we have developed in order to determine, in the absence of any suggestions by the compiler, which optimizations have been performed.

(v) Section 6 places the work described in this paper in the context of the larger translation validation project at NYU and discusses some of the outstanding research problems to be solved before a complete translation validator of an optimizing compiler can be realized.

2 Background

This section is a summary of our previous work on Tvoc. We refer the reader to [ZPFG03,ZPG$^+$03] for more details and examples.

2.1 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 executing several statements along a 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 disjuncts in the transition relation associated with this block. The first describes the `B0` to `B1` path, which is $\pi = \texttt{B0} \ \wedge \ n' = 500 \ \wedge \ y' = 0 \ \wedge \ w' = 1 \ \wedge \ n' \geq w' \ \wedge \ \pi' = \texttt{B1}$, and the second describes the `B0` to `B2` path, which is $\pi = \texttt{B0} \ \wedge \ n' = 500 \ \wedge \ y' = 0 \ \wedge \ w' = 1 \ \wedge \ n' < w' \ \wedge \ \pi' = \texttt{B2}$, 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. compatible. Note that the refinement is an equivlance relation. We use$P_T \sim S_S$ to denote that P_T is a correct translation of P_S.

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, while most loop optimizations are structure modifying (notable examples are skewing, unrolling, and peeling, that can actually be handled by both our sturcture modifying and strucutre-preserving proof approaches.)

2.2 Translation Validation of Structure-Preserving

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, Val, which is inspired by the computational induction approach ([Flo67]), originally introduced for proving properties of a single program. Rule Val (see [ZPFG03], and a variant in [ZPG$^+$03] which produces simpler verification conditions) provides a proof methodology by which one can prove that one program *refines* another. Rule This is achieved by establishing a *control mapping* from target to source locations, a *data abstraction* mapping from source variables to (possibly guarded) expressions over the target variables, and proving that these abstractions are maintained along basic execution paths of the target program.

In Val, each TS is assumed to have a *cut-point* set, i.e., 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. For each simple path, we can (automatically) construct the transition relation of the path. Typically, such a transition relation contains the condition which enables this path to be traversed and the data transformation effected by the path.

Rule Val 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.

2.3 Translation Validation of Reordering Tranformations

Structure modifying transformations are those that admit no natural mapping between the states of the source and target programs at each cutpoint. 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.

Consider the generic loop transformation in Fig. 1.

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

Fig. 1. A General Loop

Schematically, we can describe such a loop in the form "**for** $\boldsymbol{i} \in \mathcal{I}$ **by** $\prec_{\mathcal{I}}$ **do** $\mathtt{B}(\boldsymbol{i})$" where $\boldsymbol{i} = (i_1, \ldots, i_m)$ is the list of nested loop indices, and $\mathcal{I}$ is the set of the values assumed by $\boldsymbol{i}$ through the different iterations of the loop. The set $\mathcal{I}$ can be characterized by a set of linear inequalities. For example, for the loop of Fig. 1, $\mathcal{I}$ is

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

The relation $\prec_{\mathcal{I}}$ is the ordering by which the various points of $\mathcal{I}$ are traversed. For example, for the loop of Fig. 1, this ordering is the lexicographic order on $\mathcal{I}$.

In general, a loop transformation has the form:

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

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

In [ZPG+03] we propose the Rule Permute in Fig. 2. For details, soundness, and examples, see [ZPG+03].

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

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

Fig. 2. Permutation Rule Permute for reordering transformations

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. In Intel's ORC compiler, a `.l` (dot ell) file contains a description of the loop optimizations applied in the run of the optimizer. Tvoc gleans this information, verifies that indeed the optimized code follows the indicated optimization, and constrtucts the validation conditions. These conditions are then passed to the theorem prover CVC Lite [SBD02] which checks them automatically. 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.

In the next sections we address the issue of constructing the reordering transformation without the aid of the dot ell file. Another issue we address is that Rule Permute, as presetned here, only deals with transformations which reorder the execution of the entire loop's body. Some optimizations, as software pipelining and fusion/distribution, seem to fall outside the scope of the proof rule. Section 3 shows how Rule Permute can be used to handle such optimizations as well.

3 Generalization of Rule Permute

The loop transformations discussed in Section 2 cover only transformations from a single to a single loop, where there is a bijection between the iterations in one loop and the iterations in the other.

Consider a more general *loop structure*, consisting of several "simple" loops, possibly each over a different index domain, where each iteration consists of several "sub-bodies". Such a loop structure may be transformed into another loop structure. For example, in a typical loop fusion transformation, there are two simple loops (usually over the same index domain), that are transformed into a single simple loop, with each iteration consisting of two sub-bodies, one from each of the original iterations. Other transformations, such as peeling and software pipelining, can also be viewed as such loop structure transformations (we outline some examples below.)

Our thesis is that, after some pre-processing, we can view these types of transformations as an instantiations of the reordering transformation studied in Subsection 2.3, and use Rule Permute to validate them.

3.1 From Loop Structures into Simple Loops

Formally, a *loop structure* consists of:

(i) a set $\mathcal{I}_1, \ldots, \mathcal{I}_n$ of index domains;

(ii) for every $k = 1, \ldots, n$, a total ordering $\prec_{\mathcal{I}k}$ over $\mathcal{I}_k$'s elements;

(iii) for every $k = 1, \ldots, n$, number m_k and a sequence of m_k bodies $\{\mathtt{B}^k_\ell\}_{\ell=1}^{m_k}$

The code for such a loop structure is shown in Fig. 3.

$$\begin{array}{l}
\textbf{for } \boldsymbol{i} \in \mathcal{I}_1 \textbf{ by } \prec_{\mathcal{I}1} \textbf{ do } \mathtt{B}^1_1(\boldsymbol{i}); \ldots ; \mathtt{B}^1_{m_1}(\boldsymbol{i}) \\
\textbf{for } \boldsymbol{i} \in \mathcal{I}_2 \textbf{ by } \prec_{\mathcal{I}2} \textbf{ do } \mathtt{B}^2_1(\boldsymbol{i}); \ldots ; \mathtt{B}^2_{m_2}(\boldsymbol{i}) \\
\ldots \\
\textbf{for } \boldsymbol{i} \in \mathcal{I}_k \textbf{ by } \prec_{\mathcal{I}k} \textbf{ do } \mathtt{B}^k_1(\boldsymbol{i}); \ldots ; \mathtt{B}^k_{m_k}(\boldsymbol{i})
\end{array}$$

Fig. 3. An execution of a Loop Structure

Obviously, any loop of of the form of Fig. 1 in Subsection 2.3 corresponds to a single line in Fig. 3. Consider a typical loop fusion example, where the input is given by a loop structure with $k = 2$, $\mathcal{I}_1 = \mathcal{I}_2$, and $m_1 = m_2 = 1$. The fused loop is then a loop structure, with $\mathcal{J} = \mathcal{I}_1$, $m = 2$, $\mathtt{B}_1(\boldsymbol{j}) = \mathtt{B}^1$, and $\mathtt{B}_2(\boldsymbol{j}) = \mathtt{B}^1$.

We can also refer to a loop structure as a simple loop of the form

$$\textbf{for } \boldsymbol{ii} \in \mathcal{II} \textbf{ by } \prec_{\mathcal{II}} \textbf{ do } \mathtt{B}(\boldsymbol{ii})$$

by defining:

$$\mathcal{II} = \bigcup_{\ell=1}^{k} \{\ell\} \times \mathcal{I}_\ell \times \{1, \ldots, m_\ell\} \qquad \mathtt{B}(\ell_1, \boldsymbol{i}_1, t_1) = \mathtt{B}^{\ell_1}_{t_1}(\boldsymbol{i})$$

and letting $(\ell_1, \boldsymbol{i}_1, t_1) \prec_{\mathcal{II}} (\ell_2, \boldsymbol{i}_2, t_2)$ when

$$(\ell_1 < \ell_2) \;\vee\; (\ell_1 = \ell_2 \;\wedge\; \boldsymbol{i}_1 \prec_{\mathcal{I}\ell_1} \boldsymbol{i}_2) \;\vee\; ((\ell_1, \boldsymbol{i}_1) = (\ell_2, \boldsymbol{i}_2) \;\wedge\; (t_1 < t_2))$$

Thus, loop structures (as described in Fig. 3) can be converted into simple loops, and Rule Permute can be applied on transformations applied to them.

3.2 Some Frequent Transformations in between Loop Structures

We describe some of the most commonly used loop transformations from the point of view of the previous section, and show the iteration domain, ordering, and bijection F (and F^{-1} for each. See [ZPFG03] for a similar analysis of transformations from simple loops into simple loops.

Loop Fusion

The source and target for a generic loop fusion is described in Fig. 4, parts (a) and (b) respectively. For the source domain, we take $\mathcal{II} = \{1, \ldots, k\} \times \{1, \ldots, N\} \times \{1\}$, with $\prec_{\mathcal{II}}$ defined by the usual lexicographical ordering. Similarly, for the target domain, we take $\mathcal{JJ} = \{1\} \times \{1, \ldots, N\} \times \{1, \ldots, k\}$ with $\prec_{\mathcal{JJ}}$ defined by the usual lexicographical ordering. The function $F\colon \mathcal{JJ} \to \mathcal{II}$ is defined by $F(1, i, \ell) = (\ell, i, 1)$, and $F^{-1}(\ell, i, 1) = (1, i, \ell)$. Note that in order to verify Premise R3 of Rule Permute is suffices to show that for all $(1, i_1, \ell_2), (1, i_2, \ell_2) \in \mathcal{JJ}$, if $i_1 > i_2$ and $\ell_1 < \ell_2$, then

$$\mathtt{B}_{\ell_1}(i_1); \mathtt{B}_{\ell_2}(i_2) \;\sim\; \mathtt{B}_{\ell_2}(i_2); \mathtt{B}_{\ell_1}(i_1)$$

```
for  i_1=1,N
     B_1(i_1)
⋮
for  i_k=1,N
     B_k(i_k)
```

(a) Loop-1

```
for  i_1=1,N
     B_1(i_1)
     ⋮
     B_k(i_k)
```

(b) Loop-2

```
for  i=1,N
     B(i)
```

(c) Loop-3

```
for  i=1,N,k
     B(j)
     B(j+1)
     ⋮
     B(j+k-1)
```

(d) Unrolled

```
for  j_0=1,p
     B_1(j_0)
for  j_1=p+1,N_1
     B_1(j_1)
for  j_2=1,N_2
     B_2(j_2)
⋮
for  j_{k-1}=1,N_{k-1}
     B_{k-1}(j_{k-1})
for  j_k=1,q
     B_k(j_k)
for  j_{k+1}=q+1,N_k
     B_k(j_{k+1})
```

(e) Peeled

```
for  i_1=L_1,H_1
     B_1(i_1)
⋮
for  i_k=L_k,H_k
     B_k(i_k)
```

(f) Unaligned

```
for  i_1=L_1,H_1
     B_1(i_1)
⋮
for  i_r=1,H_r-L_r+1
     B_r(i_r)
⋮
for  i_k=L_k,K_k
     B_k(i_k)
```

(g) Aligned

Fig. 4. Loop Transformations

Loop Distribution

Loop distribution is the inverse of loop fusion. Thus, we can take part (b) of Fig. 4 for "Before Distribution", and part (a) of Fig. 4 for "After Distribution." The iteration domain and the ordering are just like the fusion case with the roles reversed, and so are F and F^{-1}. The verification condition remains the same.

Loop Unrolling

Generic loop unrolling is described in Fig. 4 parts (c) and (d), where we assumed k divides N. Here, the source domain is take $\mathcal{II} = \{1\} \times \{1, \ldots, N\} \times \{1\}$, the target domain is $\mathcal{JJ} = \{1\} \times \{1, k+1, \ldots, N-k+1\} \times \{1, \ldots, k\}$, and both $\prec_{\mathcal{II}}$ and $\prec_{\mathcal{JJ}}$ are usual lexicographical ordering. We then define $F(1, j, t) = (1, j+t-1, 1)$, and $F^{-1}(1, i, 1) = (1, \lfloor \frac{i}{k} \rfloor k + 1, i - \lfloor \frac{i}{k} \rfloor k)$. Note that Premise R3 of Rule $\mathsf{Permute}$ is then trivially true.

Loop Peeling

For generic loop peeling, consider the source and target in Fig. 4 parts (a) and (e) respectively. Here we take:

$$\mathcal{II} = \bigcup_{\ell=1}^{k} \{\ell\} \times \{1, \ldots, N_\ell\} times \{1\}$$

$$\mathcal{JJ} = \bigcup_{\ell=0}^{k+1} \{\ell\} \times \{1, \ldots, M(\ell)\} times \{1\}$$

$$F^{-1}(\ell, i, 1) = \begin{cases} (0, i, 1) & \ell = 1 \ \wedge \ i \leq p \\ (k+1, i, 1) & \ell = k \ \wedge \ i > q \\ (\ell, i, 1) & \text{otherwise} \end{cases}$$

where $M(0) = p; \ M(k) = 1; M(k+1) = N_k$ and for every $\ell = 1, \ldots, k-1$, $M(\ell) = N_\ell$.

Both $\prec_{\mathcal{II}}$ and $\prec_{\mathcal{JJ}}$ are usual lexicographical ordering, and, again, Premise R3 of Rule $\mathsf{Permute}$ is trivially true.

Loop Alignment

For generic loop peeling, consider the source and target in Fig. 4 parts (f) and (g) respectively. Here we take:

$$\mathcal{II} = \bigcup_{\ell=1}^{k} \{\ell\} \times \{L_\ell, \ldots, H_\ell\} \times \{1\}$$

$$\mathcal{JJ} = \bigcup_{\ell=1}^{k} \{\ell\} \times \{f_\ell(H_\ell), \ldots, f_\ell(H_\ell)\} \times \{1\}$$

$$F(\ell, i, 1) = (\ell, f_\ell^{-1}(i), 1)$$

$$F^{-1}(\ell, i, 1) = (\ell, f_\ell(i), 1)$$

where $f_r(i) = i - p + 1$, $f_r^{-1}(i) = i + p - 1$, and for every $\ell \neq r$, $f_\ell(i) = f_\ell^{-1}(i) = i$.

Both $\prec_{\mathcal{II}}$ and $\prec_{\mathcal{JJ}}$ are usual lexicographical ordering, and, again, Premise R3 of Rule Permute is trivially true.

4 Combinations

As mentioned above, a real compiler may apply *several* optimizations to transform a source program S into a target program T. In such cases, we first obtain (or guess) the sequence of individual transformations, and then synthesize a series of intermediate versions of the code, $S = I_0, I_1, \ldots, I_n = T$ based on these transformations. We then must validate the transformation from I_j to I_{j+1} for every $j = 0, \ldots, n-1$, and validate that I_n is indeed T. In this section, we illustrate this approach by assuming the sequence of transformations is given by the compiler. In the next section, we discuss heuristics for what to do if the sequence of transformations is not given.

```
for  i=0 to 99 do
     a[i+1] = x + 5
for  i=0 to 99 do
     a[i+1] = a[i+1] + a[i+2]
```

(a) Source $S = I_0$

```
a[1] := x + 5;
for  i=1 to 99 do
     a[i+1] = x + 5
     a[i] = a[i] + a[i+1]
a[100] := a[100] + a[101]
```

(b) Target $T = I_3$

Fig. 5. Input and Output

Consider the source and target program in Fig. 5 (this is an actual transformation performed by the ORC compiler). The source contains two consecutive loops over $\mathcal{I} = [0..99]$ with the usual $<$ ordering. Suppose now that the sequence of optimizations applied is alignment followed by peeling followed by fusion as shown in Fig. 6: I_1 aligns the second loop; I_2 peels the first iteration of the first loop and the last iteration of the second loop; and $I_3 = T$ fuses the loops.

To show the correctness of each of the stages, we use Rule Permute, applied to the loop structures, as described in Section 3. For example, the last (and hardest) stage of fusion, amounts to showing that

$$n_1 > n_2 \quad \Longrightarrow \quad \begin{array}{l} \mathtt{a[n_1+1] := x+5;\ a[n_2] := a[n_2] + a[n_2+1]} \ \sim \\ \mathtt{a[n_2] := a[n_2] + a[n_2+1];\ a[n_1+1] := x+5;} \end{array}$$

which can be easily established.

Typically, the compiler performs one or more loop optimizations in order to enable further global (structure preserving) optimizations. For example, the last stage I_3 in Fig. 6 can be further optimized by performing scalar replacement, loop-invariant code motion, and copy propagation to obtain the code shown in Fig. 7.

```
for  i=0 to 99 do
     a[i+1] = x + 5
for  i=1 to 100 do
     a[i] = a[i] + a[i+1]
```

(a) I_1 (Alignment)

```
a[1] := x + 5;
for  i=1 to 99 do
     a[i+1] = x + 5
for  i=1 to 99 do
     a[i] = a[i] + a[i+1]
a[100] = a[100] + a[101]
```

(b) I_2 (Peeling)

```
a[1] := x + 5;
for  i=1 to 99 do
     a[i+1] = x + 5
     a[i] = a[i] + a[i+1]
a[100] := a[100] + a[101];
```

(c) I_3 (Fusion)

Fig. 6. Stages of Optimizations

```
y := x + 5;
a[1] := y;
for  i=1 to 99 do
     a[i+1] = y
     a[i] = a[i] + y
a[100] := a[100] + a[101];
```

Fig. 7. After Structure Preserving Optimizations

This can be handled by adding one additional verification stage using the Rule Val (for structure preserving optimizations) as discussed in Section 2. Thus, the overall approach consists of three steps. First, a candidate sequence of n loop transformations is fixed. Second, intermediate representations I_1 through I_n are synthesized and the correctness of each transfomation is verified using Rule Permute. Finally, the equivalence of I_n and the target T is validated using Rule Val.

5 Heuristics

In this section, we describe techniques that we use to try to determine, in the absence of information provided by the compiler, the sequence of loop optimizations that might have been performed on the source code in order to produce target code.

Because we are still in the early stages of devising heuristics to infer the transformations that occured, we make the following simplifying (but not unreasonable) assumptions:

- We know the *mapping* from each loop structure in the source to each loop structure in the target – that is, for each loop structure in the source code, we know which loop structure in the target resulted from it. This is generally fairly easy to determine from contextual information (e.g., procedure referred to, conditional branch, variables referenced, etc.).

- We have some knowledge about the *order* in which compilers perform some transformations to enable others, since most transformations follow commonly known sequences of transformations.

5.1 General Approach

Our general approach, combining heuristics with the synthesis of intermediate versions of the code, is to apply the algorithm below to each loop structure in the source and the corresponding loop structure in the target.

Input: Source loop structure S and target loop structure T.
Output: "Valid" or "fail".
Algorithm

- $I := S$
- While $!(I \sim T)$ do
  - · $Opt := NextOpt(I, T)$; $NextOpt$ takes as input an intermediate loop structure and a target loop structure, and returns a possible next optimization that can be performed to bring the intermediate code closer to the target code, or $\perp$ if none exists; obviously, this is the part that contains the heuristics.
  - · If $Opt = \perp$, return with "failure".
  - · $I' := Opt(I)$, creating a new intermediate form resulting from the guessed optimization.
  - · Use Rules Permute and Val to establish $I' \sim I$. If validation fails, exit with "failure".

  end while
- exit with "Valid"

5.2 Criteria for Selecting the Next Optimization

The criteria upon which our selection of the next optimization are based are the following:

(i) Has the number of loops changed

(ii) Has the nesting depth of a loop changed?

(iii) Has the body size of a loop changed (i.e. have more statements ben added to the body of a loop)?

(iv) Have the bounds of a loop iteration changed?

(v) Has the use of a loop index variable changed in the body of a use (.e.g. has "i" been replaced by "i+1" in array subscripts)?

(vi) Has a non-unit step been introduced in a loop (e.g. "for i = 1 to N step k")?

These criteria are useful because different loop optimizations exhibit different combinations of the criteria. Our first attempt at a heuristic orders the tests in the order specified above, but we expect to refine this ordering through experimentation.

The loop optimizations that our tool is expected to recognize, and which of the above criteria they satisfy, are as follows.

- Peeling causes an increase in the number of loops and a change in the loop bounds of both the new loop and the existing loop.
- Alignment causes a change in the loop bounds and a constant to be added to each occurrence of the loop index variable in the body of the loop.
- Unrolling causes an increase in the size of a loop body and a non-unit step to appear in the iteration.
- Tiling causes an increase in the nesting depth of a loop, a non-unit step in the (new) outer loops, and a change in the loop bounds of the inner iterations.
- Interchange causes the order of the loop index variables in array subscripts to be changed, and possibly causes a change in the loop bounds.
- Fusion causes a decrease in the number of loops and an increase in the body size of a loop.
- Distribution causes an increase in the number of loops and an decrease in the body size of a loop.

In the next section, we provide more detail – motivated by our example from Section 3 – of the heuristic.

5.3 The $NextOpt$ Heuristic

One of the benefits of our generalized representation of a loop structure, described in Section 3, is that the representation of the source and target loops in this framework lends itself to providing clues to the transformations that occurred. Rather than give the complete definition of $NextOpt$ for all possible transformations (which we are still working on), we motivate our work here by describing how $NextOpt$ would behave on the example in Fig. 5.

The source code in Fig. 5 can be represented as the generalized loop structure

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

where

$$\mathcal{I}\mathcal{I} = (\{1\} \times \{0 \ldots 99\} \times \{1\}) \cup (\{2\} \times \{0 \ldots 99\} \times \{1\})$$

The target code in Fig. 5 can be represented by

$$\textbf{for } \boldsymbol{jj} \in \mathcal{J}\mathcal{J} \textbf{ by } \prec_{\mathcal{J}\mathcal{J}} \textbf{ do } \mathtt{B}(\boldsymbol{jj})$$

where

$$\mathcal{J}\mathcal{J} = (\{1\} \times \{0\} \times \{1\}) \cup (\{2\} \times \{1 \ldots 99\} \times \{2\}) \cup (\{3\} \times \{99\} \times \{1\})$$

(Note: we recognize a single occurrence of a loop body as a collapsed loop of one iteration – in particular, we recognized the first line of the target code as an instance of the body of the first loop of the source with $i = 0$, and the last line of the target code as an instance of the body of the second loop of the source with $i = 99$.). This representation, along with a cursory examination of the source and target loop bodies, makes the following easy to see:

(i) In $\mathcal{II}$, the last element of each triplet is 1, but in $\mathcal{JJ}$ there are triplets whose last element is 2. This indicates that the number of blocks has increased inside a simple loop within the target loop structure, something that occurs when either fusion or unrolling occurs.

(ii) Since the range of middle elements in neither $\mathcal{II}$ nor $\mathcal{JJ}$ contain a "step" (.e.g $\{1, k+1, 2k+1, ...\}$), it is unlikely that unrolling has occurred. This can also be seen by noticing that the bodies of the simple loop in the target that has increased in size (i.e. in the number of blocks within the loop) are not copies of each other, with different array indices. Thus, it is unlikely that unrolling produced this set of blocks.

(iii) The range of first elements of $\mathcal{II}$ is $\{1, 2\}$, while the range of first elements of $\mathcal{JJ}$ is $\{1, 2, 3\}$. This situation, indicating an increase in the number of simple loops in the target loop structure over the source loop structure, can only occur with peeling or distribution.

(iv) Since distribution would cause a reduction in the range of third elements of $\mathcal{JJ}$ as compared to $\mathcal{II}$ – and that has not happened in this case – it is unlikely that distribution has occurred.

(v) The range of the middle elements, i.e. the range of values of the loop index variables, in $\mathcal{JJ}$ is $\{0..100\}$ while the range of the middle elements of $\mathcal{II}$ is $\{0..99\}$. An increase in the upper bound for the loop index variables, or a decrease in the lower bound, indicates that alignment may have occurred (it could also indicate skewing, but the relationship between $\mathcal{II}$ and $\mathcal{JJ}$ when skewing occurs is substantially different than what we see here).

Once these observations are made, we are left with a set of optimizations – namely fusion, peeling, and alignment – that are candidates for being considered as the first optimization performed on the way from the source to the target.

Because alignment and peeling are considered enabling transformations for fusion, and not vice-versa, it makes sense for *NextOpt* to choose either alignment or peeling as the optimization to use in the construction of the next intermediate version of the code (this is an instance of where our knowledge of compilers is exploited in the heuristic). What this example shows is that, in many cases, either choice of alignment or peeling will lead to eventual validation.

Although the Intel ORC compiler actually performed alignment followed by peeling on this example, a heuristic choice of peeling before alignment in the validation process will work.

By choosing peeling, we will construct the next intermediate version of the code, I', to be:

```
a[1] = x + 5
```

```
for  i=1 to 99 do
     a[i+1] = x + 5
for  i=0 to 98 do
     a[i+1] = a[i+2] + 11
a[100] = a[101] + 11
```

Thus, $\mathcal{II}'$, the iteration space for I', is defined by

$$\mathcal{II}' = \{(1,0,1)\} \cup (\{2\} \times \{1 \ldots 99\} \times \{1\}) \cup (\{3\} \times \{0 \ldots 98\} \times \{1\}) \cup \{(4,99,1)\}$$

Now, comparing $\mathcal{II}'$ with $\mathcal{JJ}$, it becomes clear that alignment is required on the second loop in $\mathcal{II}'$.

6 Conclusion

This paper describes three improvements to our translation validation approach for optimizing compilers. First, we presented a generalized rule for accommodating a wider class of loop transformations. Next, we showed how combinations of optimizations can be handled by synthesizing intermediate versions of the code and validating each optimization individually. Finally, we described preliminary heuristics for guessing the sequence of optimizations, given only the source and target code.

We are currently integrating all of these improvements into our Tvoc tool. As we do so, we plan to improve our heuristics and continue using real examples to increase the power and scope of translation validation techniques.

Acknowledgement. As always, we would like to thank Amir Pnueli for many helpful discussions. Ying Hu has supplied us with examples of ORC's optimizations that were beyond the capabilities of previous versions of Tvoc.

References

[ASU88] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1988.

[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.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 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.

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

[WL91] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proc. of the SIGPLAN '91 Symp. on Programming Language Design and Implementation*, pages 33–44, 1991.

[Wol95] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.

[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.

[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 9(2)*, 2003. Preliminary version in *ENTCS*, 65(2), 2002.

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