TVOC: A Translation Validator for Optimizing Compilers

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

[1]New York University, barrett|yifang|goldberg|yinghu|amir@cs.nyu.edu
[2]University of Illinois, Chicago, lenore@cs.uic.edu

Abstract. We describe a tool called TVOC, that uses the *translation validation* approach to check the validity of compiler optimizations: for a given source program, TVOC proves the equivalence of the source code and the target code produced by running the compiler. There are two phases to the verification process: the first phase verifies loop transformations using the proof rule PERMUTE; the second phase verifies structure-preserving optimizations using the proof rule VALIDATE. Verification conditions are validated using the automatic theorem prover CVC Lite.

1 Introduction

Verifying the correctness of modern optimizing compilers is challenging because of their size, complexity, and evolution over time. *Translation Validation* [8] is a novel approach that offers an alternative to the verification of translators in general and of compilers in particular. Rather than verifying the compiler itself, one constructs a *validating tool* that, after every run of the compiler, formally confirms that the target code produced is a correct translation of the source program. A number of tools and techniques have been developed for compiler validation based on translation validation[7,8,10]. In this paper, we introduce TVOC, a tool for translation validation for compilers.

2 System Architecture

Fig. 1 shows the overall design of TVOC. TVOC accepts as input a source program S and target program T, both in the WHIRL intermediate representation, a format used by Intel's Open Research Compiler (ORC) [9] among others. Just as compilers perform optimizations in multiple passes, it is reasonable to break the validation into multiple phases, each using a different proof rule and focusing on a different set of optimizations. Currently, TVOC uses two phases to validate optimizations performed by the compiler. Below, we explain these two phases in more detail. Fig. 2 shows a program called TEST that we will use as a running example. The transformation in question is loop fusion plus the addition of an extra branch condition before the loop.

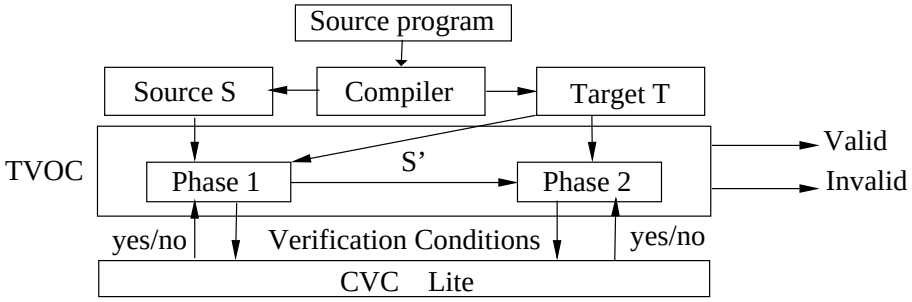

Fig. 1. The architecture of TVOC.

3 Phase 1: Reordering Transformations

In phase 1, TVOC focuses on reordering transformations. These are transformations which simply permute the order in which statements are executed, without adding or removing any statements. Examples of reordering transformations include loop interchange, loop fusion and distribution, and tiling [2]. Reordering transformations are validated using the proof rule PERMUTE, which, given a bijective function defining the permutation, produces a set of verification conditions which ensure the correctness of the transformation. Essentially, the verification conditions specify that every pair of statements whose order is exchanged by the permutation have the same result regardless of the order in which they are executed.

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

Fig. 2. S, S' and T for Program TEST

TVOC automatically determines the loop transformations by comparing the number and structure of loops in the source and target. This approach is described in [5,6] and works quite well in practice. Note that if TVOC guesses wrong, this can only lead to false negatives, never to false positives. The validation performed is always sound. For program TEST, if i_1 is a loop index variable for the first loop and i_2 is a loop index variable for the second loop, the permutation function reorders two statements exactly when $i_2 < i_1$. The verification condition can thus be expressed as follows, where $\sim$ denotes program equivalence: $(i_2 < i_1) \longrightarrow A[i_1] = 0;\ B[i_2] = 1 \ \sim\ B[i_2] = 1;\ A[i_1] = 0$. The validity of this verification condition can be checked automatically by CVC Lite [3].

Phase 1 also detects transformations such as skewing, peeling and alignment. Even though these do not actually reorder the statements, they do change the structure of the loops and so it is necessary to handle these transformations before moving on to phase 2, which assumes that S' and T have the same loop structure.

4 Phase 2: Structure-Preserving Transformations

Phase 2 handles so-called structure-preserving transformations by applying rule VALIDATE. This rule is quite versatile and can handle a wide variety of standard optimizations and transformations [1], including the insertion or deletion of statements. The main requirement is that the loop structure be the same in the source and target programs.

Two important mappings are required to apply rule VALIDATE, a *control* mapping and a *data* mapping. The control mapping is formed by finding a correspondance between a subset of locations in the target T and a subset of locations in the source S'. These locations are called *cut-points*. The sets of cut-points must include the initial and final locations in S' and T and at least one location from every loop. The other required mapping is the data mapping. Some of the source variables are identified as *observable*. These are the variables whose values must be preserved in order for a transformation to be correct. The data mapping gives a value for each observable source variable in terms of expressions over target variables. TVOC generates the control and data mappings automatically.

Fig. 3 shows program TEST annotated with cut-points. Assuming A and B are the observable variables, the data mapping simply maps A to a and B to b.

```
                                    cp0 :  if n ≥ 0 {
CP0 :  do I = 0 to N {                         do i = 0 to n {
CP1 :      A[I] = 0;                   cp1 :       a[i] = 0;
           B[I] = 1;                              b[i] = 1;
       }                                       }
CP2 :                                      }
                                           cp2 :
```

Fig. 3. Cut-points for program TEST

Validation of the source against the target is done by checking that the data mapping is preserved along every target path between a pair of cut-points. The overall correctness follows by induction [4,10]. Initially, TVOC tries to show that all variables correspond at all program locations. When it finds that the data mapping is not preserved for a given variable at some cut-point, that variable is removed from the data mapping at that location. As long as all of the observable variables are still in the data mapping at the final cut-point, the validation succeeds.

For the example, in Fig. 3, there are four possible target paths: $0 \to 1$, $0 \to 2$, $1 \to 1$ and $1 \to 2$. Therefore, four verification conditions must be checked by CVC Lite. Each verification condition checks that if the data mapping holds, and the corresponding source and target transitions are taken, then the data mapping still holds. Transitions are modeled using logical equations with *primed* variables denoting the values of variables after the transition. The verification condition for the transition from 1 to 1 is shown below:

$$A = a \ \wedge \ B = b \wedge$$
$$a' = write(a, i, 0) \wedge b' = write(b, i, 1) \wedge i' = i + 1 \wedge i + 1 \leq n \wedge n' = n \wedge$$
$$A' = write(A, I, 0) \wedge B' = write(B, I, 1) \wedge I' = I + 1 \wedge I + 1 \leq N \wedge N' = N$$
$$\rightarrow$$
$$A' = a' \ \wedge \ B' = b'.$$

In the general case, the data mapping may not be inductive, so additional invariants may be needed to establish that it holds. TVOC calculates a simple invariant for each cut-point based on data flow analysis. These invariants are often sufficient to establish the induction. Another complication is that because of branching, there may be multiple paths between two cut-points. In this case, TVOC uses the disjunction of the path transition relations. This allows TVOC for example to correctly identify a transformation in which multiple source paths are merged into a single target path.

5 Conclusions and Future Work

At this point, TVOC still has some limitations: there are some optimizations and language features that cannot yet be validated. For instance, we are still in the process of adding support for procedures and pointers.

Although TVOC has primarily been used as a research prototype and experimental platform for theoretical work, we are hoping it will be of use and interest to a broader community. In addition, we hope to receive feedback and suggestions for further improvement. We are thus making it freely available together with basic examples and documentation at http://www.cs.nyu.edu/acsys/tv/.

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 *CAV*, July 2004.
4. R. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19:19–32, 1967.
5. B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. In *COCV*, Apr. 2004.
6. Y. Hu, C. Barrett, B. Goldberg, and A. Pnueli. Validating more loop optimizations. In *COCV*, Apr. 2005.
7. G. Necula. Translation validation of an optimizing compiler. In *PLDI*, 2000.
8. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*, pages 151–166, 1998.
9. S. C. R.D.-C. Ju and C. Wu. Open research compiler (orc) for the itanium processor family. In *Micro 34*, 2001.
10. L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *FMSD*, 2005.