Translation Validation of Loop Optimizations and Software Pipelining in the TVOC Framework

In memory of Amir Pnueli

Benjamin Goldberg

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
goldberg@cs.nyu.edu

Abstract. Translation validation (TV) is the process of proving that the execution of a translator has generated an output that is a correct translation of the input. When applied to optimizing compilers, TV is used to prove that the generated target code is a correct translation of the source program being compiled. This is in contrast to verifying a compiler, i.e. ensuring that the compiler will generate correct target code for every possible source program – which is generally a far more difficult endeavor.
This paper reviews the TVOC framework developed by Amir Pnueli and his colleagues for translation validation for optimizing compilers, where the program being compiled undergoes substantional transformation for the purposes of optimization. The paper concludes with a discussion of how recent work on the TV of software pipelining by Tristan & Leroy can be incorporated into the TVOC framework.

1 Introduction

Verifying a compiler to ensure that it will produce correct target code every time it compiles a source program is a very difficult undertaking. First, compilers are large pieces of software and, given the current state of the art, verifying large pieces of software is still generally computationally intractable. Second, compilers tend to undergo updates and new releases, which would require re-verification each time.

As a proposed solution to the difficulty of verifying that a compiler will produce correct target code for any possible source program, starting in 1998 Amir Pnueli and his colleagues [10, 9, 11, 8, 12] proposed *translation validation* (TV), which is the process of verifying, for a given run of the compiler, that the target code produced during the run is a correct translation of the source program being compiled. Initially, the TV work was performed for a compiler that translated SIGNAL, a reactive language with very simple program structure (a

single outer loop), into C. This work was followed up by Pnueli and various colleagues (including this author), as well by many other researchers, who developed TV methods for industrial-strength optimizing compilers (see, e.g. [7, 2, 15, 17] among too many to list).

Performing TV for optimizing compilers is especially challenging because the optimizations performed by the compiler can significantly change the structure of a program. The TV for optimizing compilers work performed by Pnueli and colleagues resulted in a framework and implementation called TVOC [2], for Translation Validation for Optimizing Compilers, which partitions compiler optimizations into two categories:

- *Structure-preserving optimizations*: These are optimizations that do not radically change the structure of the program, so that a mapping between states of the target program and states of the source program is still possible. Examples of such optimizations include the so-called "global optimizations", such as dead-code elimination, constant folding and propagation, and common subexpression elimination.
- *Structure-modifying optimizations*: These are optimizations that radically change the structure of a program – or at least parts of the program, such as loops – so that there is no useful mapping between states of the target program and states of the source program. Examples of these optimizations include loop optimizations such as loop interchange, tiling, reversal, fusion, and distribution.

These two categories are treated differently within the TVOC framework. In both cases, based on the source and target programs and the optimizations performed, the TVOC system generates verification conditions that are then checked by a theorem prover.

The theorem prover that TVOC uses is CVC [14, 1], which is an automatic theorem prover for proving the validity of first-order formulas and has a large number of built-in theories that are useful for TV (e.g. integers, arrays, bit-vectors, etc.). The latest instantiation of CVC is CVC3 [1]. If CVC determines that the verification conditions that TVOC generates are satisfied, then the optimizations applied by the compiler were correct. Otherwise, the TVOC system indicates that the compilation was invalid.

Figure 1 shows a simple schematic of the TVOC system, as applied to the Intel Open Research Compiler a few years ago. After parsing and type checking (which the TVOC system does not validate), the compiler performs loop optimizations, global optimizations, and some machine dependent optimizations prior to code generation. Each optimization phase comprises one or more IR-to-IR transformations, taking the program in an intermediate reprentation (IR) and producing a new program represented in the same IR language. Based on these transformations, TVOC produces the set of verification conditions that are fed to the CVC theorem prover.

Figure 2 shows a slightly more detailed schematic of the TVOC system. There are separate components of TVOC for validating loop optimizations (structure

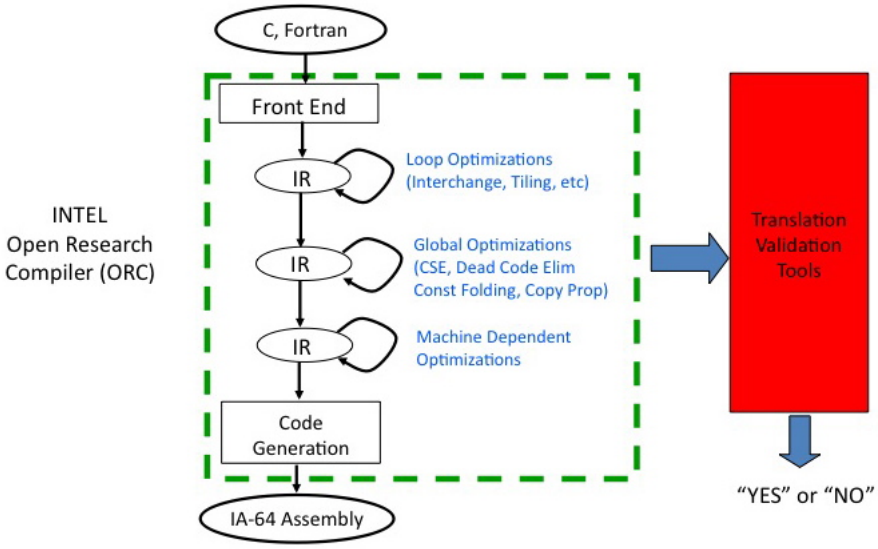


Fig. 1: A Simple Schematic of the TVOC System

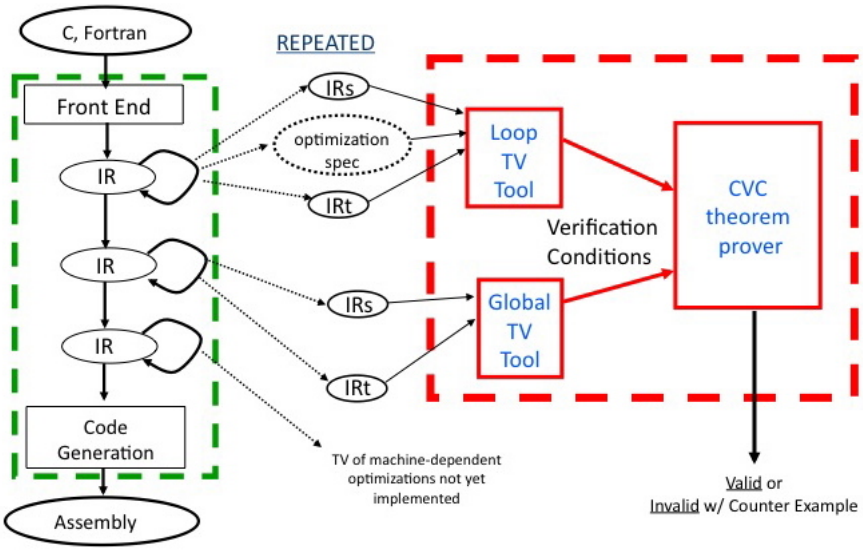


Fig. 2: Detailed Schematic of the TVOC System

modifying) and global optimizations (structure preserving). At this point, validation of machine-dependent optimizations has not been implemented in TVOC. Loop optimizations are generally performed earlier in the compilation process than global optimizations, since loop optimizations often expose opportunities for global optimization. In any case, these optimization processes tend to be iterative. The input (source) and output (target) of each optimization is fed to the appropriate module (loop TV or global TV) of TVOC, which generates the verification conditions to be fed to CVC.

We have recently begun to extend the TVOC framework, although not the implementation yet, to handle machine-dependent optimizations. One such optimization that has not been handled by TVOC, although it was addressed in other Pnueli work, is software pipelining. Recent work by Tristan & Leroy [16] for validating software pipelining using symbolic evaluation is being been adapted for the TVOC framework (i.e. using CVC). We describe here how software pipelining fits into the TVOC framework.

2 Validating global optimizations in TVOC

Global optimizations are structure preserving in the sense that they preserve the structure of a program sufficiently to permit a mapping between states of the target program (i.e. the IR representation of the program after an optimization) and states of the source (i.e. the IR representation of the program before the optimization). Although a detailed explanation of how validation of global optimizations are performed in TVOC is beyond the scope of this paper, we provide a brief description here. We refer the reader to [18] for more details and examples.

In order to validate a translation from a source program S to a target program T, where the transformations applied to S are structure-preserving, TVOC represents each program as a *transition system* [10] (TS), which is a state machine consisting of a set V of state variables, a set $\mathcal{O} \subseteq V$ of observable variables, an initial condition Θ characterizing the initial states of the system, and a transition relation ρ relating each state to its possible successors. The variables are typed, and a *state* of a TS is a type-consistent interpretation of the variables. A computation of a TS is defined to be a maximal finite or infinite sequence of states starting with a state that satisfies the initial condition such that every two consecutive states are related by the transition relation.

In order to establish that P_T, the TS representing the target program T, is a correct translation of P_S, the TS representing the source program S, we use a proof rule, Val, which is inspired by the computational induction approach [3], originally introduced for proving properties of a single program. Rule Val provides a proof methodology by which one can prove that one program *refines* another. This is achieved by establishing a *control mapping* from target to source locations, a *data abstraction* mapping from source variables to 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.

3 Validating loop optimizations

The Val rule discussed above relied on there being a mapping between the states of the source and target programs. However, there is a class of loop optimizations that optimizing compilers perform that modify the structure of loops sufficiently so that no such mapping is possible. Thus, Pnueli and his colleagues, including this author, developed and implemented in TVOC a method for validating loop optimizations that did not rely on such a mapping. We describe this method briefly here, but refer the reader to [2].

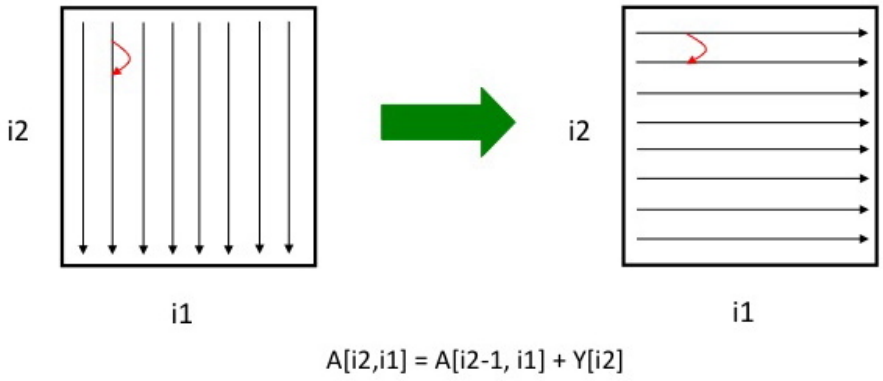
The loop optimizations that TVOC handles fall under the category of reordering transformations, which are transformations that change the order of execution of statements in the body of a loop, but do not change the number of times each statement is executed. Reordering transformations cover many of the loop optimizations performed by optimizing compilers, including fusion, distribution, reversal, interchange, and tiling.

To illustrate TVOC's validation of loop optimizations, we consider loop interchange. The loop interchange optimization reorders the nesting of a nested loop. Figure 3 shows an example of a loop interchange on a doubly-nested loop. For this example, the transformation may provide several performance benefits. First, since the expression `Y[i2]` is loop invariant in the inner loop of the transformed code, the computation of the address denoted by Y[i2] and the fetching of its value can be moved outside the inner loop. Second, if the array `A` is arranged in row major form, where adjacent elements in a row occupy consecutative locations in memory, then cache performance is likely to be improved. The illustration below shows the transformation of the access pattern over the array `A` caused by the interchange.

```
for i1 = 1 to N do
  for i2 = 2 to M do
    A[i2,i1] = A[i2-1, i1] + Y[i2]
  end
end
```

```
for i2 = 2 to M do
  for i1 = 1 to N do
    A[i2,i1] = A[i2-1, i1] + Y[i2]
  end
end
```

Fig. 3: An example of loop interchange



The curved arrows represent an execution of the assignment statement, where each element `A[i2,i1]` is assigned a value computed from the element `A[i2-1,i1]` above it. The new access pattern resulting from the optimization must preserve the relative order in which `A[i2,i1]` and `A[i2-1,i1]` are visited during execution of the loop. Otherwise, the transformation will have changed the result produced by the loop.

In order to define a single rule for validating all reordering loop transformations, we represent a loop of the form

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

by

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

where $\boldsymbol{i} = (i_1, \ldots, i_m)$ is the list of nested loop indices, $\mathcal{I}$ is the set of the values assumed by $\boldsymbol{i}$ through the different iterations of the loop, and B represents the entire body of the loop. The set $\mathcal{I}$ can be characterized by a set of linear inequalities. For example, for the above loop, $\mathcal{I}$ is defined by

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

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

In general, a loop transformation has the form:

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

Such a transformation may change the domain of the loop indices from $\mathcal{I}$ to $\mathcal{J}$, change the 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 loop body $\mathtt{B}(\boldsymbol{i})$ to the target body $\mathtt{B}(F(\boldsymbol{j}))$.

The rule used in TVOC to validate loop transformations is the Permute rule shown in Figure 4, where F is a bijection (i.e. it is one-to-one and onto) mapping iterations in the transformed loop back to iterations in the original loop.

$$\frac{\forall \boldsymbol{i}_1, \boldsymbol{i}_2 \in \mathcal{I} : \boldsymbol{i}_1 \prec_{\mathcal{I}} \boldsymbol{i}_2 \wedge F^{-1}(\boldsymbol{i}_2) \prec_{\mathcal{J}} F^{-1}(\boldsymbol{i}_1) \implies \mathtt{B}(\boldsymbol{i}_1);\mathtt{B}(\boldsymbol{i}_2) \sim \mathtt{B}(\boldsymbol{i}_2);\mathtt{B}(\boldsymbol{i}_1)}{\textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\boldsymbol{i}) \sim \textbf{for } \boldsymbol{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\boldsymbol{j}))}$$

Fig. 4: Permutation Rule Permute for reordering transformations

Intuitively, the Permute rule says that if, for any circumstance under which a reordering transformation switches the relative order of two iterations i_1 and i_2 in the source and target code, it is case that executing the body B in iteration i_1 followed by executing B in iteration i_2 is equivalent to executing B in iteration i_2 followed by executing the body in iteration i_1, then the reordering transformation is correct.

In order to apply rule Permute to a given case, it is necessary to identify the function F (and F^{-1}) and verify that the antecedent of Rule Permute is satisfied. The identification of F can be provided by the compiler, once it determines which of the relevant loop optimizations it chooses to apply. Intel's ORC compiler generates a file containing a description of the loop optimizations applied in the current phase of optimization. TVOC extracts this information (identified as "optimization spec" in Figure 2), verifies that the optimized code has resulted from the indicated optimization, and constructs the verification conditions. These conditions are then passed to CVC, which checks them automatically.

Consider the interchange example shown in Figure 3. The loop interchange transformation for that example can be characterized as follows:

$$\begin{array}{l} \textbf{for } \boldsymbol{i} \textbf{ in } \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathbf{A}[i_2 - 1,i_1] + \mathbf{Y}[i_2] \\ \Longrightarrow \\ \textbf{for } \boldsymbol{j} \textbf{ in } \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathbf{A}[j_1 - 1,j_2] + \mathbf{Y}[j_1] \end{array}$$

where

$$\mathcal{I} = \{(i_1, i_2) | 1 \leq i_1 \leq N, 1 \leq i_2 \leq M\}$$
$$\mathcal{J} = \{(j_1, j_2) | 1 \leq j_1 \leq M, 1 \leq j_2 \leq N\}$$

and $\prec_{\mathcal{I}}$ and $\prec_{\mathcal{J}}$ are lexicographic ordering on their respective iteration spaces. The functions F and F^{-1} associated with loop interchange are defined by

$$F(j_1, j_2) = (j_2, j_1)$$
$$F^{-1}(i_1, i_2) = (i_2, i_1)$$

In order to determine if loop interchange is valid on the example loop, the definitions of $\mathcal{I}$, $\mathcal{J}$, $\prec_{\mathcal{I}}$, $\prec_{\mathcal{J}}$,F, F^{-1}, and the loop body B are plugged into the antecedent of the Permute rule, namely

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

The resulting formula is then fed to CVC to determine if it is valid. If it is valid, then loop interchange optimization is correct for this example.

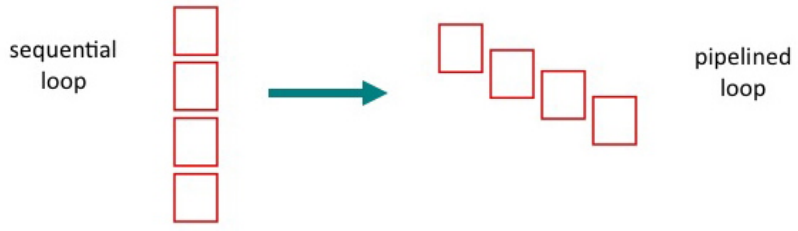
For those cases where the compiler does not indicate the loop transformations that were applied, TVOC uses a set of heuristics figure out which transformations were used.

4 Validating Software Pipelining

Machine-dependent optimizations, such as software pipelining, are not yet handled by the TVOC implementation. In this section, we discuss how TV for software pipelining can be incorporated into TVOC, based on recent work by Tristan & Leroy [16]. We start, however, with a intuitive explanation of the software pipelining optimization.

4.1 A gentle introduction to software pipelining

Software pipelining [13, 5] refers to a class of optimizations that improve program performance by overlaying iterations of a loop – essentially allowing an iteration to start before the previous iteration has completed, even if there are dependences between iterations that prohibit the iterations executing fully in parallel. Software pipelining can be view schematically as:



The benefits of software pipelining include 1) exploiting instruction-level parallelism by allowing instructions from different iterations to execute simultaneously on VLIW or superscalar machines, 2) filling delay slots in one iteration with instructions from other iterations, and 3) other improvements (register allocation, cache performance, etc.) that can be made during instruction scheduling by being able to select among instructions from several overlapping iterations.

Although software pipelining generally occurs at the instruction-scheduling phase of compilation, where the optimization is applied to machine instructions, for clarity we will show the examples in this paper in an intermediate representation (IR) that is fairly close to the source.

Consider the following simple loop:

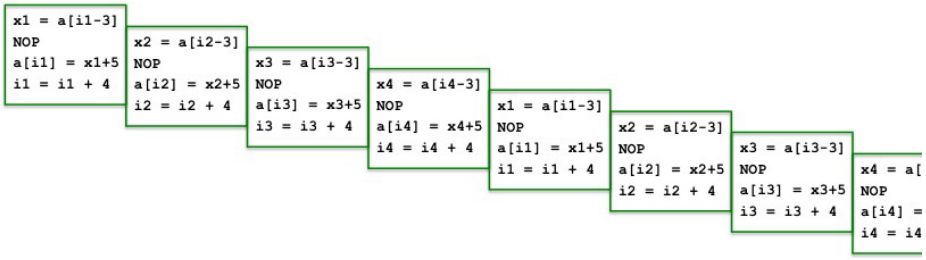
```
for i= 3 to N
  a[i] = a[i-3] + 5
```

A corresponding (high-level) intermediate representation form of the loop is:

```
i=3
while (i<=N) {
   x = a[i-3]
   NOP //delay slot
   a[i] = x+5
   i = i + 1
   }
```

We assume that the load instruction, `x = a[i-3]`, takes an extra cycle due to the memory fetch, thus a NOP ("no-op") is inserted to ensure that `x` is not referenced too early[1]. In the sequential execution of the loop, a cycle is wasted by the NOP during every iteration.

The figure below illustrates the execution of overlaid iterations in a software pipeline. These iterations continue executing as long as specified by the loop bounds.



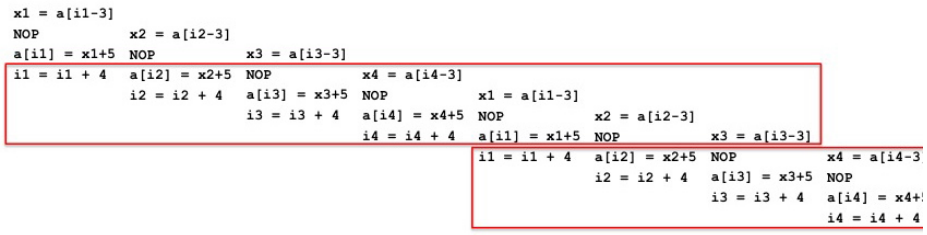
As can be seen by close examination of the above figure, the actual pipeline code is accomplished by replicating the body of the loop four times, creatng a total of four instances of the variables `i` and `x`, and then overlaying the four iterations.

[1] For simplicity, we assume a purely statically-scheduled machine with no out-of-order execution or interlocked stages.

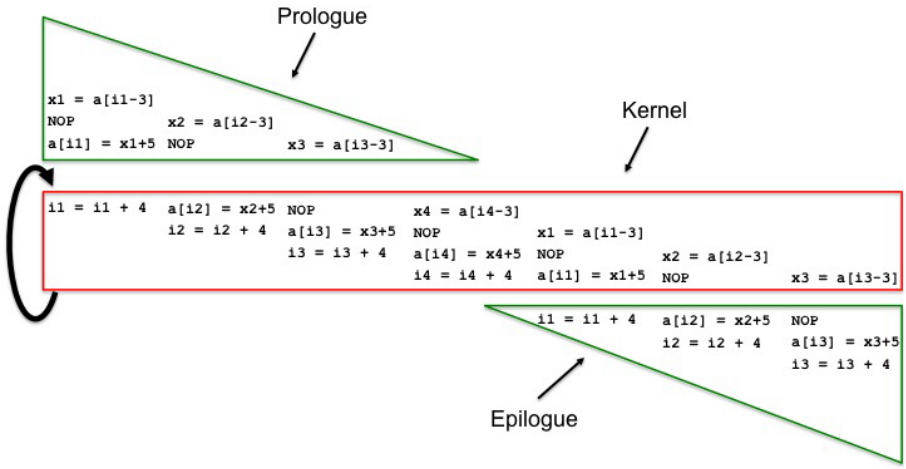
During execution, these four iterations are repeatedly executed, as implied by the figure above.

In a software pipeline, such as the one illustrated above, the instructions appearing on the same horizontal level – despite being from different iterations – can be executed simultaneously or in any order chosen by the compiler. Thus, although the NOP appears in the figure, it does not consume a cycle since there are other instructions that can be executed in that same cycle.

Upon further examination of the above figure, it can be seen that horizontal blocks of code are repeated in the execution of the overlaid iterations. This is shown in the figure below, where the code within the first large rectangle is repeated in the second rectangle (which is only partially visible) and many times subsequently.



The horizontal block of code within the large rectangle is called the "kernel" of the pipeline. Only one instance of the kernel code is actually generated, and is then executed in a loop. The figure below illustrates the repeated execution of the kernel code, preceded by a set of instructions called the "prologue" and followed by the set of instructions called the "epilogue". The prologue can be thought of as a "ramping up" of the pipeline and the epilogue as a "ramping down" of the pipeline.



For clarity, the above figure doesn't show the number of times that the kernel is executed. It can be seen from inspection that, together, the prologue and

epilogue corresponds to executing three iterations of the original loop body (note the three assignments to `x`, the three writes to `a[ ]`, etc.) and that the kernel code corresponds to four iterations of the original loop body. Thus, since the original loop executed N times, it must be the case that N is at least 3, since the prologue and epilogue will always execute once the pipelined code is entered. Furthermore, the value of $N-3$, i.e. the number of iterations of the original loop that is executed by iterating over the kernel, must be divisible by four since each iteration of the kernel corresponds to four iterations of the original loop.

Using this logic, and the notation from [16], it is clear that, in general, if the prologue and epilogue together execute μ iterations of the original loop and each iteration of the kernel executes δ iterations of the original loop, then we require that $N \geq \mu$ and that $(N - \mu)$ is a multiple of δ. To enforce these requirements, the pipeline code is generally preceded by a conditional that tests the value of N, unless N can be determined statically. If $N < \mu$, then the pipeline code will not be entered at all. If $N-\mu$ is not a multiple of δ, then the appropriate number (i.e. $(N-\mu)$ MOD δ) of iterations of the loop are peeled off and executed separately, so that the remaining iterations of the loop can be pipelined.

4.2 Validating a software pipeline

In [6], Pnueli and Leviathan described a method for validating software pipelining using an extension of the Val rule described above. This work used a mapping between transition systems resulting in a fairly complicated method.

In a recent POPL paper [16], Tristan & Leroy describe a less complicated approach, defining a simple rule to be satisfied in order to deem that the translation from the original loop into a pipeline is correct. As their paper discusses, given a source loop with a body B that is translated into the pipeline consisting of a prologue P, a kernel S, and an epilogue E, where E and P together represent μ iterations of B and S represents δ iterations of B, the translation is correct *iff*

$$B^N \sim P; S^{(N-\mu)/\delta}; E$$

That is, executing the body B of a loop N times is equivalent to executing the prologue P, followed by iterating over the kernel S for $(N - \mu)/\delta$ times, followed by the epilogue E. As discussed above, it is assumed (and enforced by other code) that $N \geq \mu$ and that $(N - \mu)$ is a multiple of δ. Tristan & Leroy noted, though, that without knowledge of N, which is a run-time value, proving the above equivalence for all possible N is very difficult. Thus, they proposed a simple rule that is sound but not complete, in that if the rule is satisfied, then the translation is correct, but there may be correct translations that do not satisfy the rule. However, their paper states that such cases don't arise in practice.

The Tristan & Leory rule can be specified as follows: Suppose a source loop whose body is B is translated into the pipeline consisting of a prologue P, a kernel S, and an epilogue E, where E and P together represent μ iterations of

B and S represents δ iterations of B. Then,

$$\frac{(B^{\mu} \sim P; E) \wedge (E; B^{\delta} \sim S; E)}{B^{N} \sim P; S^{(N-\mu)/\delta}; E} \qquad \text{(Tristan \& Leroy)}$$

where it is assumed that $N \geq \mu$ and $(N - \mu)$ is a multiple of δ.

As shown in their POPL paper, the Tristan & Leroy rule is easy to prove inductively (once a framework, such as their symbolic evaluation, is developed for reasoning about equivalence – which is not so easy). Informally, the induction proceeds as follows. Since $N - \mu$ is divisble by δ, $N = \mu + m\delta$ for some $m \geq 0$. m is used as the basis of the induction.

Base Case $m = 0$:

$$\begin{aligned} B^{\mu+m\delta} &= B^{\mu} \\ &\sim P; E \\ &= P; S^{0}; E \end{aligned}$$

Assume for any $m \leq k$, $B^{\mu+k\delta} \sim P; S^{k}; E$. Then,

$$\begin{aligned} B^{\mu+(k+1)\delta} &= B^{\mu+k\delta}; B^{\delta} \\ &\sim P; S^{k}; E; B^{\delta} \\ &\sim P; S^{k}; S; E \\ &= P; S^{k+1}; E \end{aligned}$$

Thus, for any m, $B^{\mu+m\delta} \sim P; S^{m}; E$ and since $N = \mu + m\delta$, i.e. $m = (N - u)/\delta$, $B^{N} \sim P; S^{(N-\mu)/\delta}; E$.

The intuition behind the Tristan & Leroy rule can be seen in figure 5, which is adapted (with permission) from Figure 3 in [16]. The horizontal sequence at the

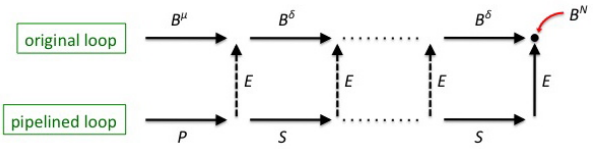


Fig. 5: Illustration of the Tristan & Leroy rule, adapted from [16]

top of the figure represents the execution of the original code and the sequence at the bottom is the execution of the pipelined code.

In their POPL paper, Tristan & Leroy describe a symbolic evaluation method for proving the equivalences $(B^{\mu} \sim P; E)$ and $(E; B^{\delta} \sim S; E)$ for a particular source loop body B and target pipeline components P, S, and E. Instead, we have incorporated the Tristan & Leroy rule into the TVOC framework, where it is used to generate two verification conditions – simply $(B^{\mu} \sim P; E)$ and $(E; B^{\delta} \sim S; E)$ – that are fed to CVC theorem prover, along with the code for B^{μ}, B^{δ}, P, S, and E. If CVC finds the two conditions valid, then pipelining is correct.

Figure 6 shows the original and pipelined loops of our example program, above, along with the verification conditions, encoded for CVC. P, S, and E in the CVC code resulted from an SSA transformation applied to the pipeline code. B^3 and B^4, corresponding to B^{μ}, B^{δ}, respectively, were generated by static loop unrolling and then an SSA transformation. Equivalence between B^3 and $P; E$ and between $E; B^4$ and $S; E$ is checked in CVC by asserting that their inputs (the initial values of `a`, the `i`'s, and the `x`'s) are equal and querying CVC about the equality of the their outputs (i.e. the final values of `a`, the `i`'s, and the `x`'s).

Software pipelining, although an optimization that can be complicated to perform, lends itself nicely to simple translation validation rules, such as the Tristan & Leroy rule, because none of the pipeline prologue, kernel, or epilogue themselves contain loops or branches. Although the compiler has freedom to rearrange instructions within each of these blocks, the resulting code will still be amenable to equivalence checking by a theorem prover.

4.3 Future Work: Validating pipelining that uses hardware support

In practice, compilers that perform software pipelining often generate code for machines, such as the Intel IA64, that provides substantial hardware support for pipelining. This hardware support includes rotating registers to provide automatic renaming of variables (such as the loop index `i` in our example above) across iterations – thus avoiding replicating identical code in overlapping iterations and reducing the size of the kernel code. Another form of hardware support for software pipelining is predication, which is the ability to turn off the execution of certain instructions at run time. Predication, in this case, supports the execution of prologue and epilogue code – which are subsets of the kernel instructions – by turning off certain instructions in the kernel during the ramp up and ramp down phases of the pipeline. As described in [4], predication can also be used to dynamically alter the software pipeline in order to preserve loop-carried dependences that can only be computed at run time.

Techniques for translation validation of software pipelining that use such hardware support have not yet been developed. As with performing TV for other kinds of machine-dependent optimizations, it will involve encoding the hardware features of the machine in a logical framework (e.g. as a set of CVC assertions).

Unrolled Source Code and Target Pipline Code

```
%B^3
REAL_ARRAY: TYPE = ARRAY INT OF REAL;
a1_b3: REAL_ARRAY;
x1_b3: REAL = a1_b3[i1_b3-3];
a2_b3: REAL_ARRAY = a1_b3 WITH [i1_b3] := x1_b3 + 5;
x2_b3: REAL = a2_b3[i1_b3-2];
a3_b3: REAL_ARRAY = a2_b3 WITH [i1_b3 + 1] := x2_b3 + 5;
x3_b3: REAL = a3_b3[i1_b3-1];
a4_b3: REAL_ARRAY = a3_b3 WITH [i1_b3 + 2] := x3_b3 + 5;
i2_b3: INT = i1_b3 + 3;
%B^4
a1_b4: REAL_ARRAY;
i1_b4: INT;
x1_b4: REAL = a1_b4[i1_b4-3];
a2_b4: REAL_ARRAY = a1_b4 WITH [i1_b4] := x1_b4 + 5;
x2_b4: REAL = a2_b4[i1_b4-2];
a3_b4: REAL_ARRAY = a2_b4 WITH [i1_b4 + 1] := x2_b4 + 5;
x3_b4: REAL = a3_b4[i1_b4-1];
a4_b4: REAL_ARRAY = a3_b4 WITH [i1_b4 + 2] := x3_b4 + 5;
x4_b4: REAL = a4_b4[i1_b4];
a5_b4: REAL_ARRAY = a4_b4 WITH [i1_b4 + 3] := x4_b4 + 5;
i2_b4: INT = i1_b4 + 4;
%PROLOGUE
i1_pl: INT;
i2_pl: INT;
i3_pl: INT;
i4_pl: INT;
ASSERT i1_pl = 2;
ASSERT i2_pl = i1_pl + 1;
ASSERT i3_pl = i1_pl + 2;
ASSERT i4_pl = i1_pl + 3;
a1_pl: REAL_ARRAY;
x1_pl: REAL = a1_pl[i1_pl-3];
x2_pl: REAL = a1_pl[i2_pl-3];
a2_pl: REAL_ARRAY = a1_pl WITH [i1_pl] := x1_pl + 5;
x3_pl: REAL = a2_pl[i3_pl-3];
%EPILOGUE
a1_ep: REAL_ARRAY;
i11_ep: INT;
i21_ep: INT;
i31_ep: INT;
i41_ep: INT;
x1_ep: REAL;
x2_ep: REAL;
x3_ep: REAL;
x4_ep: REAL;
i12_ep: INT = i11_ep + 4;
a2_ep: REAL_ARRAY = a1_ep WITH [i21_ep] := x2_ep + 5;
i22_ep: INT = i21_ep + 4;
a3_ep: REAL_ARRAY = a2_ep WITH [i31_ep] := x3_ep + 5;
i32_ep: INT = i31_ep + 4;
%EPILOGUE2, a copy of EPILOGUE
a1_ep2: REAL_ARRAY;
i11_ep2: INT;
i21_ep2: INT;
i31_ep2: INT;
i41_ep2: INT;
x2_ep2: REAL;
x3_ep2: REAL;
i12_ep2: INT = i11_ep2 + 4;
a2_ep2: REAL_ARRAY = a1_ep2 WITH [i21_ep2] := x2_ep2 + 5;
i22_ep2: INT = i21_ep2 + 4;
a3_ep2: REAL_ARRAY = a2_ep2 WITH [i31_ep2] := x3_ep2 + 5;
i32_ep2: INT = i31_ep2 + 4;
%KERNEL (S)
a1_s: REAL_ARRAY;
i11_s: INT;
i21_s: INT;
i31_s: INT;
i41_s: INT;
x21_s: REAL;
x31_s: REAL;
i12_s: INT = i11_s + 4;
a2_s: REAL_ARRAY = a1_s WITH [i21_s] := x21_s + 5;
x41_s: REAL = a2_s[i41_s - 3];
i22_s: INT = i21_s + 4;
a3_s: REAL_ARRAY = a2_s WITH [i31_s] := x31_s + 5;
x11_s: REAL = a3_s[i12_s - 3];
i32_s: INT = i31_s + 4;
a4_s: REAL_ARRAY = a3_s WITH [i41_s] := x41_s + 5;
x22_s: REAL = a4_s[i22_s - 3];
i42_s: INT = i41_s + 4;
a5_s: REAL_ARRAY = a4_s WITH [i12_s] := x11_s + 5;
x32_s: REAL = a5_s[i32_s - 3];
```

Assertions and Queries for Validation

```
%Assertions for P;E
%Connect the outputs of P to the
i1_b3: INT; %inputs of E.
ASSERT i11_ep = i1_pl;
ASSERT i21_ep = i2_pl;
ASSERT i31_ep = i3_pl;
ASSERT i41_ep = i4_pl;
ASSERT x2_ep = x2_pl;
ASSERT x3_ep = x3_pl;
ASSERT a1_ep = a2_pl;
%QUERIES FOR B^3 = P;E
%Set the inputs to B3 equal to inputs to P
ASSERT a1_b3 = a1_pl;
ASSERT i1_b3 = i1_pl;

%Query if the outputs of B^3 and E are equal
QUERY i2_b3 = i41_ep;
QUERY a3_ep = a4_b3;
QUERY x3_b3 = x3_ep;

%————————————-
%Assertions for S;E
%For S;E, the i1s, i2s, xs, and a's have to align
ASSERT i11_ep = i12_s;
ASSERT i21_ep = i22_s;
ASSERT i31_ep = i32_s;
ASSERT i41_ep = i42_s;
ASSERT x1_ep = x11_s;
ASSERT x2_ep = x22_s;
ASSERT x3_ep = x32_s;
ASSERT x4_ep = x41_s;
ASSERT a1_ep = a5_s;
%Assertions for E_2;B^4
%Need to use second copy of E, namely "_ep2"
%Align a[] output of E2 with a[] input of B^4
ASSERT a1_b4 = a3_ep2;
%Align i1 input of B^4 with i4 output of E_2
ASSERT i1_b4 = i41_ep2;
%Queries for E_2;B^4 = S;E
%Assert the equality of the inputs to E_2 and S
ASSERT a1_ep2 = a1_s;
ASSERT i11_ep2 = i11_s;
ASSERT i21_ep2 = i21_s;
ASSERT i31_ep2 = i31_s;
ASSERT i41_ep2 = i41_s;
ASSERT x2_ep2 = x21_s;
ASSERT x3_ep2 = x31_s;
%This gives the relationship among the i's in S
ASSERT i21_s = i11_s + 1;
ASSERT i31_s = i11_s + 2;
ASSERT i41_s = i11_s + 3;
ASSERT x1_ep = x11_s;
%Query if the outputs of B^4 and E are equal.
QUERY i41_ep = i2_b4;
QUERY i12_ep = i2_b4+1;
QUERY i22_ep = i2_b4 + 2;
QUERY i32_ep = i2_b4 + 3;
QUERY x1_b4 = x4_ep;
QUERY x2_b4 = x1_ep;
QUERY x3_b4 = x2_ep;
QUERY x4_b4 = x3_ep;
QUERY a5_b4 = a3_ep;
```

Fig. 6: The pipelining example in CVC

5 Conclusion

We have attempted in this paper to provide an inkling of the contribution that Amir Pnueli made to techniques for ensuring the correctness of compilers – and the extent to which his translation validation work has inspired further work in this area. A large number of papers (too many to list here, unfortunately) have been published on translation validation since Pneuli's 1998 paper, and we expect translation validation to be an important area of verification for some time.

References

1. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.
2. Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore D. Zuck. TVOC: A translation validator for optimizing compilers. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.
3. R. W. Floyd. Assigning meanings to programs. *Proc. Symp. Appl. Math*, 19:19–31, 1967.
4. Benjamin Goldberg, Emily Crutcher, Chad Huneycutt, and Krishna Palem. Software bubbles: Using predication to compensate for aliasing in software pipelines. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:211, 2002.
5. M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation (PLDI 88)*, pages 318–328, July 1988.
6. Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 280–287, New York, NY, USA, 2002. ACM.
7. George C. Necula. Translation validation for an optimizing compiler. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83–94, New York, NY, USA, 2000. ACM.
8. A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool CVT: Automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):192–201, 1998.
9. A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation for synchronous languages. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP 1998)*, volume 1443 of *Lecture Notes in Computer Science*, pages 235–246, 1998.
10. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, 1998.

11. Amir Pnueli, Ofer Shtrichman, and Michael Siegel. Translation validation: From DC+ to C*. In *International Workshop on Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 1998.
12. Amir Pnueli, Ofer Strichman, and Michael Siegel. Translation validation: From SIGNAL to C. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 231–255. Springer, 1999.
13. B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.
14. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, July 2002. Copenhagen, Denmark.
15. Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 316–326, New York, NY, USA, 2009. ACM.
16. Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–92, New York, NY, USA, 2010. ACM.
17. Anna Zaks and Amir Pnueli. CovaC: Compiler validation by program analysis of the cross-product. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2008.
18. Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3):223–247, 2003.