Validating Software Pipelining Optimizations *

Raya Leviathan
Dept. of Computer Science
Weizmann Institute of Science
raya@wisdom.weizmann.ac.il

Amir Pnueli
Dept. of Computer Science
Weizmann Institute of Science
amir@wisdom.weizmann.ac.il

ABSTRACT

The paper presents a method for translation validation of a specific optimization, software pipelining optimization, used to increase the instruction level parallelism in EPIC type of architectures. Using a methodology as in [15] to establish simulation relation between source and target based on computational induction, we describe an algorithm that automatically produces a set of decidable proof obligations. The paper also describes *SPV*, a prototype translation validator that automatically produces verification conditions for software pipelining optimizations of the SGI Pro-64 compiler. These verification conditions are further checked automatically by the CVC [12] checker.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*; D.2.4 [**software Engineering**]: Software/Program Verification—*Formal methods, Validation*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Pipeline processors*

General Terms

Verification

Keywords

Compilers, Optimization, Pipeline processors, Verification, Translation Validation

1. INTRODUCTION

There is a growing awareness of the importance of formally proving the correctness of safety-critical portion of software systems. However, proving the correctness of the implementation written in a high level language is not sufficient. It should also be verified that whatever correctness has been achieved on the higher level is not impaired in the translation done by the compiler. But, formally verifying s full fledge optimizing compiler is not feasible due to its size, evolution over time and possibly, proprietary consideration. The translation validation approach offers an alternative to the verification of translators in general and of compilers in particular. The efficacy of this approach were demonstrated by the discovery of a bug in the Trimaran compiler, which may produce an illegal memory access in the generated code (see [15],[14]).

Modern architectures rely on the compiler to extract maximum performance and to ensure correct utilization of the processor. These requirements from the compiler are inevitable when considering compiler for an EPIC (Explicitly Parallel Instruction Computation) processor. In this work we choose to address the problem of formally verifying software pipelining optimization of an EPIC processor.

We extend the framework described in [15] and [14] to handle machine dependent optimizations, which have an important role for processors of the EPIC family. Here, we describe our method to validate software pipelining optimization. The method is implemented for the code generator of the SGI Pro-64 compiler (to be called Pro-64), which produces code for the Intel-IA64 architecture. Our tool, *SPV*, *Software Pipeline Validation*, runs fully automatically, starting with source and translated programs and exiting with an either **Valid** or **Invalid** answer.

The formalization is based on *transition semantics* [9]. The correctness definition is based on *refinement relation between transition systems* [3]. Our tool produces the proof obligations needed to establish the correctness of the translation. These proof obligations belong to the decidable logics of Pressburger Arithmetic, arrays and uninterpreted functions. As such, they can be checked by a fully automatic tool. For most of the applications reported here, we used the CVC (*Cooperating Validity Checker*) tool [12] in order to dispatch the proof obligations. The average run time to validate a translation which includes software pipelining optimization,

*This research was done as part of the SafeAir project of the Eurtopean Commission

CASES 2002, October 8–11, 2002, Grenoble, France.

measured on programs containing small loops, is less than a second.

2. RELATED WORK

To the best of our knowledge, no other work can directly validate software pipelining optimization. The work of Necula [10] which aims towards validation of a rich class of optimizations does not try to validate this particular optimization. The method used there applies various heuristics to recover and identify the optimizations performed and the associated refinement mapping. It is based on a special-purpose verification engine included in the tool. Whereas, we are using the method used by VOC that is described in [15]. However, it is extended to handle a specific optimization typical to EPIC processors. While VOC [14] handles machine independent optimizations which are done in an early stage of the compiler, our research concentrates on the code generation stage which, for EPIC processors such as the IA64 involves aggressive optimizations. VOC uses STeP[2] as the verification engine, and thus the proof can not be done automatically.

Works that do handle the code generation part of a compiler are those based on the Verifix project as described in [13] and [6], yet they can handle a limited type of optimizations and code selection. In particular they do not address software pipelining optimization. The verification engine used in the Verifix project is the verification tool PVS, but usually the proofs are automatic.

Another approach, described in [8] requires a real execution of the code. This limitation may not be acceptable in many embedded systems.

The work described in [5] does handle optimizations which are close to loop pipelining in the sense that it addresses a parallel DSP processor. It uses the checker approach, and requires a very strong relation between the compiler and the checker.

3. TRANSITION SYSTEMS

As common formal semantics for both source and target systems, we introduce *Transition Systems* $\mathcal{S}$, a variant of the transition systems of [9]. A *Transition System* $\mathcal{S} =< V, \mathcal{O}, \Theta, \rho >$ is given by the following components:

- A finite set V of *system variables*, The variables in V are typed, and a *state* of a $\mathcal{S}$ is a type-consistent interpretation of the variables. One of the variables is the control variable which represents the location of the control of the program. For a state s and a variable $x \in V$, we denote by $s[x]$ the value that s assigns to x.

- $\mathcal{O} \subseteq V$ a set *observable variables*. The observable variables are the variables which are visible to the external world.

- Θ an *initial condition* characterizing the initial states of the system.

- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to its successor $s' \in \Sigma$ by referring to both unprimed and primed versions of the state variables. The transition relation $\rho(V, V')$ identifies state s' as a *successor* of state s if $\langle s, s' \rangle \models \rho(V, V')$, where $\langle s, s' \rangle$ is the joint interpretation which interprets $x \in V$ as $s[x]$, and x' as $s'[x]$. Thus, for example, 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.

A *computation* of a transition system $\mathcal{S}$ is a maximal finite or infinite sequence of states $\sigma : s_0, s_1, ...$, starting with a state that satisfies the initial condition and, every two consecutive states are related by a transition relation, i.e. $\exists \tau :< s_i, s_{i+1} >\models \rho_\tau$ for every i, $0 \leqq i + 1 <| \sigma |$.
Let $\mathcal{S}_\mathcal{S} = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $\mathcal{S}_\mathcal{T} = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be two transition systems, to which we refer as the *source* and *target* $\mathcal{S}$'s, respectively. We denote by $X \in \mathcal{O}_S$ and $x \in \mathcal{O}_T$ the corresponding observable variables in the two systems. We say that $\mathcal{S}_\mathcal{T}$ is a *correct translation* (*refinement*) of $\mathcal{S}_\mathcal{S}$ if for every finite (i.e., terminating) P_T-computation $\sigma^T : s_0^T, \ldots, s_m^T$, there exists a finite P_S-computation $\sigma^S : s_0^S, \ldots, s_k^S$, such that $s_m^T[x] = s_k^S[X]$ for every $X \in \mathcal{O}_S$.

4. BLOCK PROGRAMS

Block programs are programs constructed out of *blocks* of *instructions*. Where *instructions* are the legal machine instructions of the underlying physical processor. An instruction changes the state of the processor by assigning new values to the processor memory and registers. We divide the group of possible instructions to: branch instructions, which are either conditional instruction (*branch* < condition> < label>), or unconditional one (*branch*<label>), and regular instructions. A *block* is a sequence of *instructions* with no branch, except, possibly, the last instruction of the block. The <label> in a branch instruction is the branch *destination*, and its value is the index of a block. A branch destination starts a block.

A block program P is constructed out of *blocks*. Let $P = \{B_0, ...B_n\}$ be a *program*, then B_0 is the *entry* block. Each block B of a program may have one or two *successors,* denoted by *succ*(B) . If the last instruction of B_i is regular then *succ*$(B_i) = \{B_{i+1}\}$. If the last instruction is a unconditional branch with destination B_j, then *succ*$(B_i) = \{B_j\}$. If the last instruction is a conditional branch with destination B_j, then *succ*$(B_i) = \{B_j, B_{i+1}\}$.

Each machine code program P, corresponds to a *transition system* $S^P =< V, \mathcal{O}, \Theta, \rho >$ given by the following components:

- $V = \{v_1, ..., v_n\} \cup \{pc\}$ is a finite set of system *variables*. where $pc \in \{0, ..., n + 1\}$ is the control variable. Its value denotes the index of the block in

which control currently resides. When $pc = n+1$, we say that the program is terminated, or reached the *exit* point. Block index (i) and block name (B_i) are used interchangeably.

- $\mathcal{O}$: The observable variables are memory variables, function parameters and return value.
- Θ : The *initial condition* assertion where, $\Theta \rightarrow pc = 0$.
- ρ : The *transition relation*.

Each block B_i in the program is represented by a *block transition* relation $\rho_i(V, V') = (pc = i \bigwedge_{v_i \in V} v_i' = Exp(V))$. The program transition relation is : $\rho = \bigvee_{i=0}^{n} \rho_i(V, V')$. In writing transition relation, we often use the notation $Pres(U)$ for $U \subseteq V$ which is an abbreviation for $\bigwedge_{v \in U} (v' = v)$, stating that all U-variables are preserved by the relevant transition.

5. THE VALIDATE PROCEDURE

We assume that both the source and target programs can be presented as a combination of basic blocks, each of which containing a sequence of assignments and optionally terminated by a branch.

Consider, for example, the source and target programs presented in Fig. 1. Their presentation as a directed graph, whose nodes are blocks and the edges connect a block to its successors, is given in Fig. 2. We present a proof rule, called VALIDATE, which will enable us to prove that a target program correctly implements a given source program.

```
0 :  S := 0;                    0 :  s := 0;
     I := 1;                         i := 1;
1 :  do                         1 :  do
{                               {
     S := S + 2 * A[I];              s := s + a[i];
                                     s := s + a[i];
     I + +;                          i + +;
}    while(I <= 100)            }    while(i <= 100)
2 :                             2 :
```

- Source- -Target -

Figure 1: Source and Target Programs

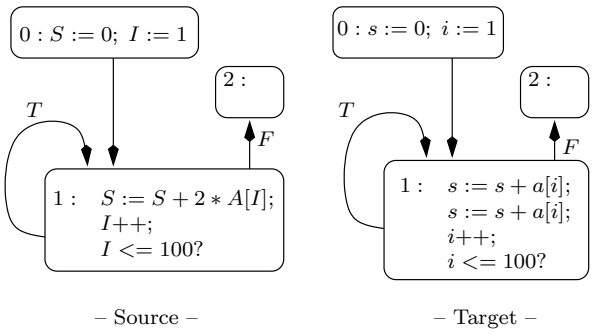

Figure 2: Example source and target as directed basic blocks graph

1. Construct a control abstraction $\kappa : \{0, .., n+1\} \mapsto \{0, .., m+1\}$ such that

$$\kappa(0) = 0 \wedge \kappa(n+1) = m+1.$$

2. For each block, B_i form an invariant φ_i that states a target property which should hold true whenever $pc = B_i$.

3. Construct an abstraction mapping

$$\alpha : PC = \kappa(pc) \wedge (p_1 \rightarrow V_1 = e_1) \wedge \cdots \wedge (p_k \rightarrow V_n = e_k),$$

where, for each $i = 1, \ldots, k$, p_i is a condition depending on the target variables (including pc), $V_i \neq pc$ is a source variable, and e_i is an expression over the target variables. It is required that when $pc \in \{1, n+1\}$, α implies $x = X$ for each observable variable $X \in \mathcal{O}_{\mathcal{S}}$.

4. For each $i \in \{0, .., n\}$ construct a verification condition

$$C_i : \left(\begin{array}{l} \varphi_i \wedge \alpha \wedge \alpha' \wedge \rho_i^T \rightarrow \\ \quad \bigvee_{j \in succ(B_i)} (pc' = j \wedge \varphi_j') \wedge (\rho^S_{\kappa(i)} \vee V_S' = V_S) \end{array} \right)$$

The disjunct $V_S' = V_S$ allows the execution of some target blocks to be mapped on an empty source execution, which modifies no source variables.

5. Establish the validity of the verification conditions.

We illustrate the application of rule VALIDATE to establish that the target program of Fig. 1 correctly implements the source program in that figure. To do so, we choose

$\kappa(i)$:	i for $i \in \{0, 1, 2\}$
α :	$PC = pc \wedge S = s \wedge I = i \wedge A = a$
φ_i :	1 (*true*) for every $i = 0, 1, 2$
$Succ(0)$:	$\{1\}$
$Succ(1)$:	$\{1, 2\}$

These choices lead to the following two verification conditions:

$$C_0 : \left(\begin{array}{l} \cdots \wedge \underbrace{PC' = pc' \wedge S' = s' \wedge I' = i' \wedge A' = a'}_{\alpha'} \quad \wedge \\ \underbrace{s' = 0 \wedge i' = 0 \wedge pc' = 1}_{\rho_0^T} \quad \rightarrow \\ \qquad\qquad \underbrace{S' = 0 \wedge I' = 0 \wedge PC' = 1}_{\rho_0^S} \end{array} \right)$$

$$C_1 : \left(\begin{array}{l} PC = pc \ \wedge \ S = s \ \wedge \ I = i \ \wedge \ A = a \ \wedge \\ \left(\begin{array}{ll} pc = 1 \wedge a' = A' & \wedge \\ pc' = if \ \ (i' \leq 100) \ \ 1 \ \ else \ \ 2 & \wedge \\ s' = s + a[i] + a[i] & \wedge \ \ i' = i + 1 \end{array} \right) \ \wedge \\ PC' = pc' \ \wedge \ S' = s' \ \wedge \ I' = i' \ \wedge \ A' = a' \\ \quad \rightarrow \quad (pc' = 1 \ \vee \ pc' = 2) \ \wedge \\ \left(\begin{array}{l} PC' = if \ \ (I' \leq 100) \ \ 1 \ \ else \ \ 2 \ \wedge \\ S' = s + 2 * A[I] \wedge I' = I + 1 \wedge PC = 1 \end{array} \right) \end{array} \right)$$

6. ARCHITECTURE DEFINITION

In this section, we define the syntax of a pseudo machine code, VIR - *virtual intermediate representation*, for a virtual architecture, based on the IA-64 family of architectures. To simplify the presentation, we prefer not to use the original machine-language syntax, but will present machine programs in a higher-level, C-like, programming style. However, the actual implementation of our approach works directly on the genuine IA-64 machine language syntax.

6.1 Machine Registers

The program refers to the following registers:

- r_i - A general integer register.
- lc - Loop count register. A special register used to count the iterations within a loop. It also controls the loop completion.
- t_i - An integer register in the rotating register file.
- p_i - A one bit register, called *predicate register* in the rotating predicate register file. $\vec{p}$ is the vector of the predicate registers.

6.2 Machine Operations

Assigning a value to a memory variable represents a *store-to-memory* operation, while the occurrence of a memory variable on the right side of an assignment, stands for a *memory-load* operation. Thus $a := t_1$ stands for storing the value of register t_1 into memory variable a. Direct assignment of a memory variable to another memory variable is not allowed, since this operation is not supported by the underlying machine. Arithmetic operations are used with their usual meaning. We use *wait* to indicate no operation. Usually, this is used when the CPU waits for the completion of an operation, such as a *load-from-memory*.

A special syntax represents the rotation of the register file: $\langle t_n := t_{n-1} := \cdots := t_i \rangle$ with the following semantics: $t_n{}' = t_{n-1} \wedge \cdots \wedge t_2{}' = t_1$. The equivalent notation is used for the predicate rotating register file. The predicated execution feature of the machine is expressed by using an *"if* (p_{c_k})" prefix. Before starting an N iteration loop, with Sc pipeline stages, the predicate array p is initialized to 10^{Sc-1}. This means that the first predicate register, is initiated to 1, and the remaining $Sc-1$ predicate registers are initiated to 0. All statements appearing on a single command line are executed in parallel within a single step.

7. SOFTWARE PIPELINING

Software pipelining is a technique that takes advantage of advanced architecture features such as parallelism (multiple memory and arithmetic units), rotating register file, predicate register and special branch instructions [1, 7]. Software pipelining increases a loop throughput by overlapping the loop's iterations; that is, by initiating successive iterations before prior iterations complete, and achieving saturation of functional units. To pipeline a loop, the compiler should find an instruction schedule that best utilizes the functional units, to achieve minimal execution time, yet without causing a register jam.

One technique for loop scheduling, is *Modulo Scheduling* [11]. To find an overlapped schedule, the compiler must take into account the constraints imposed by the availability of functional units and registers. Suppose that execution of one iteration takes C cycles. Consideration of the data dependencies within the loop body and instructions latency leads to a calculation of an *initiation interval* - Int, which is the number of instruction cycles issued for iteration i, before iteration $i+1$ can be initiated. The loop body is divided into *stages* whose execution time is Int cycles. The number of stages is $Sc = C/Int$. Let O_1, O_2 be two operations using the same machine resource which are scheduled to cycles number c_1 and c_2 of a loop body, respectively. Then, it is required that ($c_1 \mod Int$) should be different from ($c_2 \mod Int$). When all such constraints are satisfied, we have a sound *modulo schedule*.

EXAMPLE 1.

Consider the C program in Fig. 3. In this loop, there

```
int a[100], b[100], N;
main(){
        int I = 0;
        do
        {       a[I] = b[I] + 5;
                I++;
        } while (I < N);
}
```

Figure 3: C source

is no data dependency between iterations. The target program expressed in VIR, without any optimization appears in Fig. 4 The compiler calculates an Int of 1 cycle for this loop, but since the *load* delay is 2 cycles, one iteration time (i.e. C) is 4 cycles. The number of stages, Sc, is thus 4. Pipelining can be achieved, by initiating source iteration *i+1* one cycle after iteration i. Executing the resulting instruction scheduling is demonstrated in Fig. 5 for the same loop body, with $N = 6$. ld stands for load, w for wait, st for store and add for add, $op(i)$ stands for execution instruction op for source iteration i. For example $ld(2)$ stands for the load operation of iteration number 2. In the target program, the optimizing compiler produces a new loop, the

```
i1 := 0; i2 := 0; lc := 0;
do
{
    t1 := b[i1]; i1 := i1 + 1;                    –Load
    wait one cycle for the load delay to expire;  –Wait
    t2 := t1 + 5;                                 –Add
    a[i2] := t2; i2 := i2 + 1;                    –Store
    lc + +;
}while (lc < N);
```

Figure 4: Unoptimized target code

cycle						
1	*ld(1)*					
2	*w(1)*	*ld(2)*				
3	*add(1)*	*w(2)*	*ld(3)*			
4	*st(1)*	*add(2)*	*w(3)*	*ld(4)*		
5		*st(2)*	*add(3)*	*w(4)*	*ld(5)*	
6			*st(3)*	*add(4)*	*w(5)*	*ld(6)*
7				*st(4)*	*add(5)*	*w(6)*
8					*st(5)*	*add(6)*
9						*st(6)*

Figure 5: Loop pipeline schedule

target loop, whose body is composed of the operations of the pipeline when it is in its steady state, as are cycles 4,5,6. This operations are also called the loop *kernel*. Cycles 1,2,3 are the loop *prolog*, while cycles 7,8,9 are the loop *epilogue*. One iteration is completed at each cycle. The number of target iterations is *N+Sc-1*= 9.

After allocating registers, using the rotating register file, the target code is the one presented in Fig. 6, or equivalently, by the IA-64 assembly code in Fig. 11 of the appendix.

```
p1 := 1; p2 := 0; p3 := 0; p4 := 0;
lc := 0; i1 := 0; i2 := 0;
do{
l0 :
        if (p4) a[i1] := t2; i1 := i1 + 1;   Stage 4
        if (p3) t1 := t4 + 5;                Stage 3
        if (p1) t2 := b[i2]; i2 := i2 + 1;   Stage 1
        ⟨t4 := t3 := t2 := t1⟩;          Register rotation
        ⟨p4 := p3 := p2 := p1⟩;          Predicate rotation
        lc++;
        if (lc < N) p1 := 1 else p1 := 0;
while (lc < N + 3);
l1 :
```

Figure 6: Target pipelined loop

8. VALIDATION OF SOFTWARE PIPELINING OPTIMIZATION

In this section we describe a method to validate software pipelining loop optimization. This method needs only a small number of heuristics, which are based on information printed by the compiler upon user request. In many cases software pipeline optimization is preceded by a loop unrolling pass. A method for proving the validity of loop unrolling transformation is described in [15].

8.1 A General Software Pipelining Representation

Consider a general C loop as presented on the left side of Fig. 7, and its target pipelined code presented on the right side of this figure. Sc is the number of stages the optimizing compiler chose for this loop, s_j is a list of operations whose execution is predicated by the value of p_{c_i}, $t_1, \ldots, t_l$ are the rotating registers used for this loop, and $p_1, \ldots p_{Sc}$ are the predicate registers.

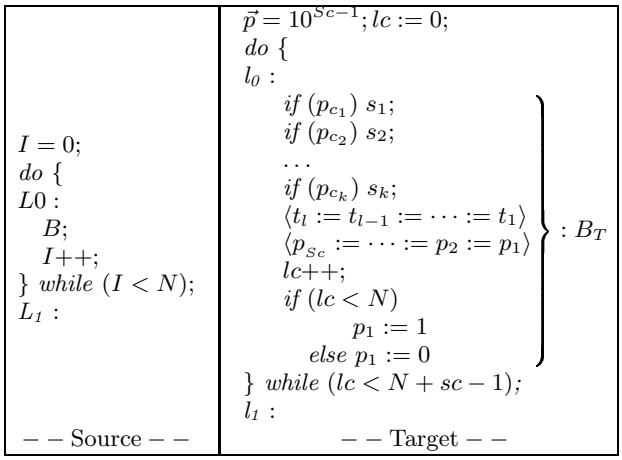

Figure 7: General form of a pipelined loop

All s_j which depend on the same predicate register, belong to the same pipeline stage. $c_i = c_j$ is possible. Also $\forall j \in [1..k] : 1 \leq c_j \leq Sc$. B_T is the body of the pipelined loop, as shown in Fig. 7. The number of target iterations is $N + Sc - 1$. Note that the first and last $Sc - 1$ iterations of the loop execute only some of the stages contained in the loop's body since some of the p_i's are 0.

8.2 The General Idea

Let ρ^S and ρ^T stand for the transition relations representing the loop body of the source and target systems respectively, and α be the abstraction mapping. We want to compute an invariant φ which can be used in the procedure VALIDATE. We first note that while the source loop iterates N times, the target loop iterates $N + Sc - 1$ times. We handle this by choosing the idle source transition to emulate the first $Sc-1$ target iteration. Next we want to compute the invariant as required in step 2 of the VALIDATE rule.

In the following subsections, we describe a method for the automatic computation of the invariant φ.

8.2.1 Computing φ

Rather than employ a single invariant, we distinguish between the versions of the invariant which apply to the various prolog, epilog, and stead-state iterations. Thus, we generate different versions of the invariant φ_i for $i = 0, \ldots, Sc-2, N, N+1, \ldots, N+Sc-2$, and a common version φ_{st} corresponding to all steady-state iterations which span the range $Sc-1 \leq i < N$. Thus, independently of the value of N, we will always deal with $2 \cdot Sc - 1$ different versions of the invariant.

In order to compute the different versions of the invariant, we use a restriction of the notion of *symbolic evaluation* and *symbolic state* as defined in [10]. A *symbolic state* is an assertion of the form

$$\varphi : \bigwedge v_i = e_i,$$

where $v_i \in V$ are target system variables, and e_i are expressions.

DEFINITION 2. *Symbolic Evaluation- Let V be a set of variables, φ be an assertion describing a symbolic state, and let ρ be a transition relation. The symbolic state resulting from the application of the transition ρ to the symbolic state described by φ (also known as the* postcondition *of φ relative to ρ) is given by:*

$$\varphi \circ \rho \quad \triangleq \quad \exists V^- : (\varphi(V^-) \ \wedge \ \rho(V^-, V),$$

where V^- is another copy of the variables V, intended to capture their values before the transition is taken.

The algorithm in Fig. 8 successively computes the $2 \cdot Sc - 2$ different cases of the invariant φ_i, by symbolically applying the transition ρ_B corresponding to a single execution of the loop's body B_T to the previous symbolic state. The assertion φ_0 is computed based on the initiation phase before the loop starts (see Section 9). We then proceed to compute $\varphi_1, \ldots, \varphi_{Sc-2}$ as the invariants which hold at the beginning of each prolog iteration. The assertion φ_{st} is valid for the loop steady state while the assertions $\varphi_N, \ldots, \varphi_{N+Sc-2}$ are valid at the beginning of the corresponding epilog iterations.

$\varphi_0 := \varphi := \mathit{Init} \ \wedge \ \vec{p} = 10^{Sc-1}$
for $(i := 1;\ i \leq Sc-2;\ i := i+1)$
  $\{\varphi_i := \varphi := (\varphi \ \wedge \ lc+1 < N) \circ \rho_B\}$
$\varphi_{st} := \varphi := (\varphi \ \wedge \ lc+1 < N) \circ \rho_B$
for $(i := 0;\ i \leq Sc-2;\ i := i+1)$
  $\{\varphi_{N+i} := \varphi := (\varphi \ \wedge \ lc+1 \geq N) \circ \rho_B\}$

Figure 8: Algorithm for computing φ

The condition $lc+1 < N$ added to φ in the computation of $\{\varphi_1, \ldots, \varphi_{Sc-2}, \varphi_{st}\}$ guarantees that the new value of p_1 will always be 1. Similarly, the condition $lc+1 \geq N$ appearing in the computation of the cases $\{\varphi_N, \ldots, \varphi_{N+Sc-2}\}$ guarantees that the new value of p_1 will be 0.

Following the VALIDATE procedure, the verification condition to be validated is:

$$C_0 : \left(\begin{array}{l} \varphi \wedge \alpha \wedge \alpha' \wedge \rho_B \rightarrow \\ \ (pc' = 0 \wedge \varphi' \ \vee \ pc' = 1) \wedge (\rho^S_{L0} \vee Pres(V^S)) \end{array} \right)$$

However, since when computing the invariant, we already split it into $2 \cdot Sc - 1$ different cases, it is necessary to generate a similar number of verification conditions. These conditions after appropriate simplification are listed in Fig. 9.

8.3 Example

We illustrate this algorithm on the running example shown in Fig. 3 (the source) and in Fig. 6 (the target). The symbolic evaluation process of φ_i is listed in Fig. 10, while the verification condition for the steady state, as produced by SPV, is listed below.

Let $\alpha : lc >= 3 \ \wedge \ I = lc - 3 \vee lc < 3 \ \wedge \ I = 0$ and $\kappa : PC = pc$.

The verification condition for the loop steady state is:
$\alpha : I = lc - 3 \wedge A = a \wedge B = b \wedge$
$\alpha' : I' = lc' - 3 \wedge A' = a' \wedge B' = b' \wedge$
$\varphi_{st} \wedge 3 \leqq lc < N \wedge$

$$\rho_B : \left\{ \begin{array}{l} a' = a \ with([i1] : t_2) \wedge \\ i'_1 = i_1 + 1 \wedge i'_2 = i_2 + 1 \wedge t'_3 = b[i_2] \wedge \\ t'_2 = t_4 + 5 \wedge t'_1 = t_4 + 5 \wedge t'_4 = t_3 \wedge \end{array} \right.$$

$$lc' = lc + 1 \wedge PC = pc \wedge PC' = pc' \wedge pc' = 0$$

$$\rightarrow \rho_{L_0} : \left\{ \begin{array}{l} A' = A \ with([I] : B[I] + 5) \wedge \\ I' = I + 1 \wedge \\ (I + 1 < N) \wedge 3 \leqq lc' \leqq N \\ \wedge PC' = \mathit{if} \ (I + 1 < N) \\ \qquad\qquad \mathit{then} \ 0 \ \mathit{else} \ 1 \end{array} \right\} \ \bigwedge \varphi'_{st}$$

The SPV tool produces a set of equivalent verification conditions, which are automatically verifyed by CVC.

9. SPV: VALIDATOR OF SOFTWARE PIPELINING OPTIMIZATION

In this section we give an overview of SPV tool. We intend to run the tool for each code-generator pass separately. It should be noted that the code generator of Pro-64 has about 15 passes, some of which are performing the same type of optimization. Currently, we validate the software pipelining stage, and produce the verification conditions for pipelined loops.

From CGIR to a Transition System. The tool inputs, source and target systems, use syntax and semantics of the *code generator internal language - CGIR* of the Pro-64. The source program is the CGIR before running software pipelining optimization and the target is the pipelined code. CGIR represents the program as a list of pseudo machine instructions, but also includes annotations that describe the program as a graph. The nodes of the graph are blocks, with at most one branch instruction, which is the last instruction of the block. The edges connect each block to its predecessors and successors. For each block SPV produces the *compressed transition* of the block, composed of multiple assignment. At most one assignment exists for each variable. The CGIR annotations are used by SPV as hints. The *expression* class (*expression.cxx, expression.h)* enables the efficient production of compressed

using: $\mu \triangleq \alpha \wedge \alpha'$

Prolog verification conditions-

$$
\begin{array}{ll}
VC_0: & \mu \wedge lc = 0 \wedge \varphi_0 \wedge \rho_B \rightarrow lc' = 1 \wedge \varphi'_1 \wedge Pres(V^S) \\
VC_1: & \mu \wedge lc = 1 \wedge \varphi_1 \wedge \rho_B \rightarrow lc' = 2 \wedge \varphi'_2 \wedge Pres(V^S) \\
\vdots & \\
VC_{Sc-2}: & \mu \wedge lc = Sc-2 \wedge \varphi_{Sc-2} \wedge \rho_B \rightarrow lc' = Sc-1 \wedge \varphi'_{st} \wedge Pres(V^S)
\end{array}
$$

Steady state verification conditions-

$$
\begin{array}{ll}
VC^a_{st}: & \mu \wedge Sc-1 \leqq lc < N-1 \wedge \varphi_{st} \wedge \rho_B \rightarrow Sc-1 \leqq lc' < N \wedge \varphi'_{st} \wedge \rho^S_{L_o} \\
VC^b_{st}: & \mu \wedge lc = N-1 \wedge \varphi_{st} \wedge \rho_B \rightarrow lc' = N \wedge \varphi'_N \wedge \rho^S_{L_0}
\end{array}
$$

Epilogue verification conditions-

$$
\begin{array}{ll}
VC_N: & \mu \wedge lc = N \wedge \varphi_N \wedge \rho_B \rightarrow lc' = N+1 \wedge \varphi'_{N+1} \wedge \rho^S_{L_0} \\
VC_{N+1}: & \mu \wedge lc = N+1 \wedge \varphi_{N+1} \wedge \rho_B \rightarrow lc' = N+2 \wedge \varphi'_{N+2} \wedge \rho^S_{L_0} \\
\vdots & \\
VC_{N+Sc-2}: & \mu \wedge lc = N+Sc-2 \wedge \varphi_{N+Sc-2} \wedge \rho_B \rightarrow lc' = N+Sc-1 \wedge \ \wedge pc' = 1 \wedge \rho^S_{L_o}
\end{array}
$$

Figure 9: Verification conditions for pipelined loop ($\mu \triangleq \alpha \wedge \alpha'$)

φ_0 :
$i_2 = lc \wedge i_1 = lc \wedge \vec{p} = (0,0,0,1)$

φ_1 :
$(\varphi_0 \wedge lc+1 < N) \circ \rho_B \sim$
$t_3 = b[lc-1] \wedge i_2 = lc \wedge i_1 = lc-1 \wedge \vec{p} = (0,0,1,1)$

φ_2 :
$(\varphi_1 \wedge lc+1 < N) \circ \rho_B \sim$
$t_4 = b[lc-2] \wedge i_2 = lc \wedge i_1 = lc-2 \wedge t_3 = b[lc-1] \wedge \vec{p} = (0,1,1,1)$

φ_{st} :
$(\varphi_2 \wedge lc+1 < N) \circ \rho_B \sim$
$t_3 = b[lc-1] \wedge t_2 = b[lc-3]+5 \wedge t_4 = b[lc-2] \wedge i_1 = lc-3 \wedge i_2 = lc \wedge \vec{p} = (1,1,1,1)$

φ_N :
$(\varphi_{st} \wedge lc+1 \geq N) \circ \rho_B \sim$
$t_3 = b[lc-1] \wedge t_2 = b[lc-3]+5 \wedge t_4 = b[lc-2] \wedge i_1 = lc-3 \wedge \vec{p} = (1,1,1,0)$

φ_{N+1} :
$(\varphi_N \wedge lc+1 \geq N) \circ \rho_B \sim$
$t_2 = b[lc-3]+5 \wedge t_4 = b[lc-2] \wedge i_1 = lc-3 \wedge i_2 = lc-1 \wedge t3 = b[lc-2] \wedge \vec{p} = (1,1,0,0)$

φ_{N+2} :
$(\varphi_{N+1} \wedge lc+1 \geq N) \circ \rho_B \sim$
$t_2 = b[lc-3]+5 \wedge i_1 = lc-3 \wedge i_2 = lc-2 \wedge t3 = b[lc-3] \wedge t_4 = b[lc-3] \wedge \vec{p} = (1,0,0,0)$

Figure 10: Example - Symbolic Evaluation of φ

transitions, as well as substitution and some simplification. This class supports basic arithmetic and logic operations, *ITE - (if then else)* operation as well as array *lookup* and *update*.

Producing the control abstraction. We use the annotations that describe the control flow, to produce the control abstraction. Usually, block B_i in the source system is mapped to a block with the same index in the target system.

Identifying Loop Linear Inductive Variables. The current preliminary version of the tool produces the initial invariant of the software pipeline loop, φ_0, by using the compiler annotations. In particular, registers and temporary variables which point to arrays are annotated by the code generator. In the future versions we will construct these invariants algorithmically (see [9] page 207). This construction is performed for both the source and the target system.

Producing the Abstraction Mapping. Mainly based on the code generator annotations.

Constructing φ for Software Pipelining Loops (*swp.cxx, swp.h)*. SPV computes the assertions for the prolog, epilogue and steady state following the algorithm describes in subsection 6.2.1. It uses the expression class to substitute and simplify expressions. A special simplification is performed by substituting constant p_i values for the different pipeline stages. SPV outputs 2*sc-1 assertions to be proved. The tool is able to communicate with more than one verification engine (currently ICS [4] and CVC).

Validation Using CVC. The produced verification conditions are all in the decidable logics of Pressburger Arithmetic, arrays and uninterpreted functions. The tool output is a set of verification conditions which are fed into CVC. CVC checks the conditions and outputs either **Valid** or **Invalid**. It is a matter of less than a second, for CVC, to validate all verification conditions of the running example of this paper.

10. REFERENCES

[1] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):368–432, September 1995.

[2] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, M. P. H. B. Simpa, and T. Uribe. STEP *The Stanford Temporal Prover Educational Release*. Computer Science Department, Stanford University, July 1998.

[3] K. Engelhardt, W.-P. de Roever, et al. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1999.

[4] J.-C. Filliâtre, SamOwre, H. Rueß, and N. Shanka. ICS: Integrated canonizer and solver. In *Proc. 13th Intl. Conference on Computer Aided Verification (CAV'01)*, 2001.

[5] S. Glenser, R. Geiβ, and B. Boesler. Verified code generation for embedded systems. In *Compiler Optimization meets Compiler Verification*, pages 23–40, 2002.

[6] G. Goos and W. Zimmermann. Verification of compilers. In B. Steffen and E. R. Olderog, editors, *Correct System Design*, volume 1710, pages 201–230. Springer, Nov 1999.

[7] R. Huff. Lifetime-sensitive modulu scheduling. In *Programming Language Design and Implementation. SIGPLAN*, 1993.

[8] C. Jaramillo, R. Gupta, and M. Soffa. Debugging and testing optimizers through comparision checking. In *Compiler Optimization meets Compiler Verification*, pages 87–103, 2002.

[9] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[10] G. C. Necula. Translation validation for an optimizing compiler. In A. Press, editor, *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, pages 83–95, 2000.

[11] B. Rau, M. Schlansker, and P. Tirumalai. Code generation schemas for modulo scheduling loops. In *Proc. 25th annual international symposium on microarchitectur*, pages 158–169, 1992.

[12] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[13] W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM-Approach. *Journal of Universal Computer Science*, 3(5):504–567, May 1997.

[14] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A translation validation for optimizing compilers. In *Compiler Optimization meets Compiler Verification*, pages 6–22, 2002.

[15] L. Zuck, A. Pnueli, and R. Leviathan. Validation of optimizing compilers. Technical Report MCS01-12, Weizmann Institute of Science, 2001.

APPENDIX

A. PRO64 TARGET CODE

$L1:$

$(p19)$	*st*	$[r3] = r35, 4$	//[Stage4]
$(p18)$	*add*	$r34 = 5, r37$	//[Stage3]
$(p18)$	*nop*		//[Stage3]
$(p16)$	*ld*	$r35 = [r2], 4$	//[Stage1]
$(p16)$	*nop*		//[Stage1]
	br.ctop	$L1;;$	

Figure 11: IA-64 assembly code

The assembly code of the running example is listed in Fig. 11. $r3, r2$ are general purpose register, $r34, r35, r37$ are in the rotating register file, and $p16, p18, p19$ are predicate registers. *br.ctop* is a special branch instruction that updates and checks the loop count, rotates the registers, and update the predicate registers.