Validating Software Pipelining Optimizations

Raya Leviathan and Amir Pnueli
Dept. of Computer Science, Weizmann Institute of Sciences*

August 22, 2001

Abstract

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of safety-critical components of systems. Most formal verification methods verify the correctness of a high-level representation of the system against a given specification. However, if one wishes to infer from such a verification the correctness of the code which runs on the actual target architecture, it is essential to prove that the high-level representation is correctly implemented at the lower level. That is, it is essential to verify the correctness of the translation from the high-level source-code representation to the object code, a translation which is typically performed by a compiler (or a code generator in case the source is a specification rather than a programming language).

Formally verifying a full-fledged optimizing compiler, as one would verify any other large program, is not feasible due to its size, ongoing evolution and modification, and, possibly, proprietary considerations. The translation validation method used in this paper is a novel approach that offers an alternative to the verification of translators in general and compilers in particular. According to the translation validation approach, rather than verifying the compiler itself, one constructs a validation tool which, after every run of the compiler, formally confirms that the target code produced on that run is a correct translation of the source program.

The paper presents a method for translation validation of a specific optimization used to increase the instruction level parallelism in EPIC type of architectures. Based on our general methodology to establish simulation relation between source and target based on computational induction, we describe an algorithm that automatically produces assertions that help in this process.

1 Introduction

There is a growing awareness, in both industry and academia, of the crucial role of formally proving the correctness of safety-critical systems, or portions thereof. Most verification methods deal with the high-level specification of the system. However, if one is to prove that

*e-mail: raya@wisdom.weizmann.ac.il
the high-level specification is correctly implemented at the lower level, one needs to verify
the compiler which performs the translation. Verifying the correctness of modern optimizing
compilers is challenging due to the complexity of the target architectures as well as the
sophisticated analysis and optimization algorithms used in the compilers.

Formally verifying a fully fledged optimizing compiler, as one would verify any other large
program, is not feasible due to its size, evolution over time and possibly, proprietary consider-
ations. Translation validation is a novel approach that offers an alternative to the verification
of translators in general and of compilers in particular. According to the translation vali-
dation approach, rather than verifying the compiler itself, one constructs a validating tool
which, after every run of the compiler, formally confirms that the target code produced is a
correct translation of the source program.

The introduction of new classes of microprocessor architectures, such as the EPIC (Ex-
plicitly Parallel Instruction Computing) class exemplified by the Intel IA-64 architecture, led
to a new family of sophisticated optimizations, currently being developed and incorporat-
edy into architecture-targeted compilers such as the Trimaran and the SGI PRO64.

Our ultimate goal is to develop a methodology for the translation validation of advanced
optimizing compilers, with an emphasis on EPIC-targeted compilers. Though our method-
ology offers a solution to a wide range of optimization, in this particular paper we choose to
concentrate on one of the code generator optimizations called software pipelining.

Here we introduce the theory of a correct translation. This theory provides both a precise
definition of the notion of a target program being a correct translation of a source program,
and the methods by which such a relation can be formally established. We then apply this
methodology to the software pipelining optimization phase of the case study compiler.

2 Transition Systems

As common formal semantics for both source and target systems, we introduce Transition Systems
TS's, a variant of the transition systems of [SSP98]. A Transition System \( S = (V, O, \Theta, \rho) \) is
a state machine consisting of \( V \) a set of state variables, \( O \subseteq V \) a set of observable variables, \( \Theta \)
an initial condition characterizing the initial states of the system, and \( \rho \) 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 primed versions refer to the values of the variables
in the successor states, while unprimed versions refer to their value in the pre-transition
state. 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.

The observable variables are the variables we care about. When comparing two systems,
we require that the observable variables in the two systems match. Typically, we require that
the output file of a program, i.e. the list of values printed through its execution, be identified
as observable variables. If desired, the history of external procedure calls of a selected set of
procedures can also be included among the observable variables.
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, i.e., \( s_0 \models \Theta \), and every two consecutive states are related by the transitions relation, i.e. \( \langle s_i, s_{i+1} \rangle \models \rho \) for every \( i, 0 \leq i + 1 < |\sigma| \).

Let \( P_s = (V_s, O_s, \Theta_s, \rho_s) \) and \( P_t = (V_t, O_t, \Theta_t, \rho_t) \) be two TS's, to which we refer as the source and target TS's, respectively. Such two 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 \( X \in O_s \) and \( x \in O_t \) the corresponding observables in the two systems. We say that \( P_t \) is a correct translation (refinement) of \( P_s \) if for every finite (i.e., terminating) \( P_t \)-computation \( \sigma^T : \sigma^T_0, \ldots, \sigma^T_m \), there exists a finite \( P_s \)-computation \( \sigma^S : \sigma^S_0, \ldots, \sigma^S_k \), such that 
\[
s^T_m[x] = s^S_k[X] \quad \text{for every } x \in V_t.
\]

### 2.1 The VALIDATE procedure

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

Consider, for example, the source and target program presented in Fig. 1.

\[
\begin{align*}
S & := 0; \\
\text{for}(i := 1; \ I <= 100; \ I++) & \quad \text{for}(i := 1; \ i <= 100; \ i++) \\
\{ \ S := S + 2 \cdot A[I] \} & \quad \{ \ s := s + a[i]; \ s := s + a[i]; \}
\end{align*}
\]

--- Source ---

--- Target ---

Figure 1: A Source program and its target

Their presentation as a directed graph whose edges are basic blocks 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.

1. Establish a control abstraction \( \kappa \) mapping target locations (such as 0, 1, 2 in Fig. 2) into corresponding source locations. The control abstraction should map the initial and terminal target locations (0 and 2, respectively) into the initial and terminal source locations.

2. For each target location \( i \), form an invariant \( \varphi_i \) that states a property which should hold true whenever control visits location \( i \).

3. Establish a data abstraction

\[
\alpha : (p_1 \rightarrow v_1 = E_1) \land \cdots \land (p_n \rightarrow v_n = E_n)
\]

assigning to some source state variables \( v_i \in V_s - \{ \pi \} \) an expression \( E_i \) over the target state variables, conditional on the (target) boolean expression \( p_i \). Note, that \( \alpha \) may contain more than one clause for the same variable.

\footnote{\(|\sigma|\), the length of \( \sigma \), is the number of states in \( \sigma \). When \( \sigma \) is infinite, its length is \( \omega \).}
4. For each pair of target locations $i$ and $j$ such that $j$ is a successor of $i$ in the target program graph, form the verification condition:

$$C_{ij} : \varphi_i \land \alpha \land \rho^T_{ij} \land \alpha' \rightarrow (\rho^S_{\kappa(i),\kappa(j)} \lor \kappa(j) = \kappa(i) \land V^j_S = V^j_S) \land \varphi'_j$$

In this formula, $\rho^T_{ij}$ represents the transformation effected by the target basic block connecting location $i$ to $j$. Similarly, $\rho^S_{\kappa(i),\kappa(j)}$ represents the transformation effected by the source basic block connecting location $\kappa(i)$ to location $\kappa(j)$. Note that in the case that $\kappa(j) = \kappa(i)$ we allow the target transition to be emulated by a source idling step in which the source program does not change its state.

5. Establish the validity of all the generated verification conditions.

We will illustrate the application of rule \textsc{validate} to establish that the target program of Fig. 1 correctly implements the source program in that figure. To do so, we choose

$$\kappa(i) : \quad i \quad \text{for } i \in \{0, 1, 2\}$$

$$\alpha : \quad S = s \land I = i$$

$$\varphi_i : \quad 1 \quad \text{for every } i = 0, 1, 2$$

Figure 2: Example source and target as directed basic blocks graph
These choices lead to the following three verification conditions:

\[
\begin{align*}
C_{01} : & \quad \cdots \land s' = 0 \land i' = 1 \land \cdots \land S' = s' \land I' = i' \\
& \quad \implies S' = 0 \land I' = 1 \\
C_{11} : & \quad \left( S = s \land I = i \right) \land \left( i \leq 100 \land s' = s + A[i] + A[i] \right) \land \left( i' = i + 1 \right) \\
& \quad \implies \left[ \begin{array}{l}
I \leq 100 \\
S' = S + 2 \times A[I] \\
I' = I + 1
\end{array} \right. \\
C_{12} : & \quad \left( S = s \land I = i \right) \land \left( i > 100 \land s' = s \right) \land \left( i' = i \right) \\
& \quad \implies \left[ \begin{array}{l}
I > 100 \\
S' = S \\
I' = I
\end{array} \right.
\end{align*}
\]

It is not too difficult to see that these three verification conditions are valid.

Next, we consider a more complicated example, which requires the use of non-trivial invariants \( \varphi_i \). Here, the source program is the same as in Fig. 1 but the target program is given by the version presented in Fig. 3.

\[
s := 0; \quad t := A[1]; \\
\quad \text{for} \{ i := 1; \ i \leq 100; \ i++ \} \\
\quad \{ s = s + A[i] + t; \ \text{if} \ i < 100 \ \text{then} \ t := A[i + 1] \}
\]

Figure 3: A more complex target program

We choose

\[
\begin{align*}
kappa(k) : & \quad k \ \text{for} \ k \in \{0,1,2\} \\
\alpha : & \quad I = i \land S = s \\
\varphi_0 = \varphi_2 : & \quad 1 \\
\varphi_1 : & \quad (i < 100) \rightarrow t = A[i]
\end{align*}
\]

3 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 programming style. However, the actual implementation of our approach works directly on the genuine IA-64 syntax.
3.1 Machine Registers

Our program refer to the following registers:

- \( r_i \) - A general integer register.
- \( l_c \) - 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.
- \( p[i] \) - An array of one bit registers, called predicate registers.

In the actual architecture, there is a fixed number of predicate registers and, like the \( t_i \)'s, they are rotated after every loop's iteration. However, in our presentation here we prefer to treat them as an array whose size equals the number of iterations, and which are referenced as \( p[l_c + c_k] \) for various constants \( c_k \)'s.

3.2 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. A special syntax represents the rotation of the register file: \( \langle t_n := t_{n-1} := \cdots := t_i \rangle \) with the following semantics: \( t_n' = t_{n-1} \land \cdots \land t_2' = t_1 \).

The predicated execution feature of the machine is expressed by using an \textbf{if} \( p[l_c + c_k] \) \textbf{then} statement. Before starting an \( N \) iteration loop, with \( sc \) pipeline stages, the predicate array \( p \) is initialized to \( 0^{sc-1}10^{sc-1} \). This means that the first \( sc-1 \) and the last \( sc-1 \) entries of the array \( p \) are set to 0, while the middle \( N \) entries are set to 1. We use \textit{wait} to indicate no operation. Usually, this is used when we wait for the completion of an operation, such as a load-from-memory.

3.3 Parallel Execution

All statements appearing on a single command line are executed in parallel within a single step. However, in case a register is referenced by more than one statement on the same command line, we should interpret the overall effect as though the statements were executed in a sequential order from left to right. For example, the effect of a line containing the statements:

\[
t_2 := t_1; \quad t_1 := 1,
\]

is to store in \( t_2 \) the old value of \( t_1 \), while loading the constant 1 into \( t_1 \).
4 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 [AJLA95, Huf93]. Software pipelining increases a loop's 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 [RST92]. 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 \( cn \) cycles. Considering data dependencies of the loop body and instructions latency, leads to calculating the \( II - \text{initiation interval} \), 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 \( II \) cycles. The number of stages is \( sc = cn/II \). 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 \( (c_1 \mod II) \) should be different from \( (c_2 \mod II) \). When all such constraints are satisfied, we have a sound modulo schedule.

Example 1 Consider the C program in Fig. 4. In this loop, there is no data dependency

\[
\text{int } a[100], b[100], N \\
\text{main()}
\]

\[
\text{int } i; \\
\text{for}(i=0; i<\text{N}; i++)
\]

\[
\text{a}[i] = \text{b}[i] + 5; \\
\}
\]

Figure 4: C source

between iterations. The target program expressed in VIR, without any optimization appears in Fig. 5. The compiler calculates an \( II \) of 1 cycle for this loop, but since the load delay is

```plaintext
\[ i_1 := 0; i_2 := 0; \\
\text{for}(\text{lc} = 0; \text{lc} < \text{N}; \text{lc} := \text{lc} + 1) \{
\text{t}_1 := b[i_1]; i_1 := i_1 + 1; \quad \text{Load} \\
\text{wait one cycle for the load delay to expire; \quad \text{Wait} \\
\text{t}_2 := t_1 + 5; \quad \text{Add} \\
\text{a}[i_2] := t_2; i_2 := i_2 + 1; \quad \text{Store}
\}
```

Figure 5: Unoptimized target code
2 cycles, one iteration time \( cn \) 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. 6 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,

\[
\begin{array}{c|ccc}
\text{cycle}  & ld(1) & w(1) & ld(2) \\
\text{cycle} 2 & add(1) & w(2) & ld(3) \\
\text{cycle} 3 & st(1) & add(2) & w(4) \\
\text{cycle} 4 & st(2) & add(3) & w(5) \\
\text{cycle} 5 & st(3) & add(4) & w(6) \\
\text{cycle} 6 & st(4) & add(5) & ld(7) \\
\text{cycle} 7 & st(5) & add(6) & \\
\text{cycle} 8 & & & \\
\text{cycle} 9 & & & \\
\end{array}
\]

Figure 6: Loop pipeline schedule

the 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 ramping up cycles, prolog, while cycles 7,8,9 are the ramping down, epilog. 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. 7, or equivalently, by the IA-64 assembly code in Fig. 10 of the appendix.

\[
p := 0^31^N0^3;\\for(\text{lc} := 0; \text{lc} < N + 3; \text{lc} := \text{lc} + 1)\{\\    \text{if} (p[\text{lc}]) \text{ then } a[i_1] := t_2; i_1 := i_1 + 1; \quad \text{--Stage 4} \\
    \text{if} (p[\text{lc}+1]) \text{ then } t_1 := t_1 + 5; \quad \text{--Stage 3} \\
    \text{if} (p[\text{lc}+3]) \text{ then } t_2 := b[i_2]; i_2 := i_2 + 1; \quad \text{--Stage 1} \\
\}\\t_4 := t_3 := t_2 := t_1; \quad \text{--Register rotation}
\]

Figure 7: Target pipelined loop

5 Validation of Software Pipelining Optimization

In this section we describe a method to validate software pipelining loop optimization. This method needs only a minor use of heuristics, which are based on information printed by the compiler upon user request. In many cases software pipeline optimization is preceded by loop unrolling pass. The method of proving the validity of loop unrolling transformation is described in [ZPL01].
5.1 A General Software Pipelining Representation

Consider a C loop of the general form as in the left side of Fig. 8, and its target pipelined code presented on the right side of this figure. $sc$ is the number of stages the optimizing compiler chooses for this loop, $s_j$ is one or more operations whose execution conditionally depends on the value of $p[lc + c_j]$. $t_1, \ldots, t_t$ are the rotating registers used for this loop.

$$
\begin{align*}
\text{for}(I := 0; \ I < N; \ I++) & \\
\{B\} & \\
\text{-- Source --} & \\
\text{-- Target --}
\end{align*}
$$

Figure 8: 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 $\forall j \in [1..k]: \ 0 \leq c_j < sc$. $B_T$ is the body of the pipelined loop, as shown in Fig. 8. The number of target iterations is $N + sc - 1$. We are interested in the special iterations of the prolog and epilog. In these iterations not all the instructions in the loop body are executed, due to the values of the corresponding predicate registers. We get the list of machine instructions that are executed in these iterations, by deleting the instructions whose predicate register value is 0.

We define $B_T(i)$ for $i \in [1..sc - 1]$ and $i \in [N + 1..N + sc - 2]$:

$B_T(i)$ - The list of instructions that are executed at iteration $i \in [1..sc - 1]$ of the loop (a prolog iteration).

$B_T(N + i)$ - The list of instructions that are executed at iteration $N + i : i \in [1..sc - 2]$ of the loop (an epilog iteration)

For example, we introduce $B_T(i)$ of the program in Fig. 7. In the figure, instructions are marked by their pipeline stage number.

Prolog iterations:

$B_T(1)$: stage 1; register rotation.

$B_T(2)$: stage 1; (empty stage 2); register rotation.

$B_T(3)$: stage 3; stage 1; (empty stage 2); register rotation.

Epilog iterations:

$B_T(N + 1)$: stage 4; stage 3; (empty stage 2); register rotation

$B_T(N + 2)$: stage 4; stage 3; register rotation

9
5.2 The General Idea

Let $\rho^S$ and $\rho^T$ stand for the transition relations representing the loop body of the source and target systems respectively, and $\alpha$ be the data abstraction. We want to produce an invariant $\varphi$, to 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 produce the invariant $\varphi$ such that $\alpha \land \rho^T \land \varphi \land \alpha' \rightarrow \rho^S \land \varphi'$.

In the following subsections, we describe a method for producing the $\varphi$ invariants.

5.2.1 Producing $\varphi$

Upon completion of iteration $i + sc - 1$ of the target system, the target system state is equivalent to the source system state after completion of iteration $i$ and parts of iterations $i + 1, \ldots, i + sc - 1$. For example iteration $i + 1$ of the source is done up to stage $sc - 1$. The invariant assertion $\varphi$ should express this fact, as well as hiding target observable variables that are changed before those of the source system. We use the notion of symbolic evaluation and symbolic evaluation state as defined in [Nec00], but in a more restricted way.

**Definition 1** Symbolic State - Is an assertion $\varphi$, of the form $\varphi : \bigwedge v_i = \text{Exp}(V)$, where $v_i \in V$ are target system variables.

**Definition 2** Symbolic Evaluation - Let $v_i \in V$ be a set of variables, and $v_i^- \in V^-$ denote the values of these variables before the transition is taken. The symbolic state resulting from the application of an assignment $[x := \text{Exp}(V)]$ to the symbolic state described by $\varphi$ is given by:

$$\varphi \circ [x := \text{Exp}(V)] \triangleq \exists V^- : \varphi^- \land x = \text{Exp}(V^-)$$

We use the operator $\circ$ to denote the symbolic evaluation. The definition can easily be extended to a list of assignments.

The algorithm in Fig. 9 computes $2 \times sc - 3$ different assertions, by symbolically applying the program $[B_T(i); l_c := l_c + 1]$ to the previous state. Given that $\varphi_0$ is valid before the first iteration, one of each $\varphi_1, \ldots, \varphi_{sc-1}$ is valid at the end of one prolog iteration. The assertion $\varphi_{sc-1}$ is valid for the loop steady state while the assertions $\varphi_{N + 1}, \ldots, \varphi_{N + sc - 2}$ are valid at the end of the corresponding epilog iterations.
\[\text{for } (i := 1; i \leq sc - 1; i := i + 1) \{ \]
\[\varphi_i : \varphi_{i-1} \circ [B_T(i); \ lc := lc + 1] \]
\[
\}
\[\varphi_N := \varphi_{sc-1};\]
\[\text{for } (i := 1; i \leq sc - 2; i := i + 1) \{ \]
\[\varphi_{N+i} : \varphi_{N+i-1} \circ [B_T(N + i); \ lc := lc + 1] \]
\[
\}\]
\[\varphi : \]
\[lc = 0 \rightarrow \varphi_0 \land \]
\[lc = 1 \rightarrow \varphi_1 \land \]
\[
\vdots \]
\[sc - 1 \leq lc \leq N \rightarrow \varphi_{sc-1} \land \]
\[lc = N + 1 \rightarrow \varphi_{N+1} \land \]
\[lc = N + 2 \rightarrow \varphi_{N+2} \land \]
\[
\vdots \]
\[lc = N + sc - 2 \rightarrow \varphi_{N+sc-2} \]

Figure 9: Algorithm for computing \( \varphi \)

We illustrate this algorithm on the running example:

\( \varphi_0 : \)
\[i_0 = lc \land i_1 = lc \]
\( \varphi_1 : \)
\[\varphi_0 \circ [t_2 := b[i_2]; \ i_2 := i_2 + 1; \ \langle t_4 := t_3 := t_2 := t_1 \rangle; \ lc := lc + 1] = \]
\[t_3 = b[lc - 1] \land i_2 = lc \land i_1 = lc - 1 \]
\( \varphi_2 : \)
\[\varphi_1 \circ [t_2 = b[i_2]; \ i_2 := i_2 + 1; \ \langle t_4 := t_3 := t_2 := t_1 \rangle; \ lc := lc + 1] = \]
\[t_4 = b[lc - 2] \land i_3 = lc \land i_1 = lc - 2 \land t_3 = b[lc - 1] \]
\( \varphi_3 : \)
\[\varphi_2 \circ [t_1 := t_4 + 5; \ t_2 := b[i_2]; \ i_2 := i_2 + 1; \ \langle t_4 := t_3 := t_2 := t_1 \rangle; \ lc := lc + 1] = \]
\[t_3 = b[lc - 1] \land t_2 = b[lc - 3] + 5 \land t_4 = b[lc - 2] \land i_1 = lc - 3 \land i_2 = lc \]
\( \varphi_N : \)
\( \varphi_3 \)
\( \varphi_{N+1} : \)
\[\varphi_N \circ [i_1 := i_1 + 1; t_1 := t_4 + 5; \ \langle t_4 := t_3 := t_2 := t_1 \rangle; \ lc := lc + 1] = \]
\[t_2 = b[lc - 3] + 5 \land i_1 = lc - 3 \]
\( \varphi_{N+2} : \)
\[\varphi_{N+1} \circ [i_1 := i_1 + 1; t_1 := t_4 + 5(t_4 := t_3 := t_2 := t_1); \ lc := lc + 1] = \]
\[t_2 = b[lc - 3] + 5 \land i_1 = lc - 3 \]

We conclude with:

\( \varphi : \)
\[\forall j \in \{0,1,2\} \ lc = j \rightarrow \varphi_j \land \]
\[(lc \geq 3) \land (lc \leq N) \rightarrow \varphi_3) \land \]
\[(lc = N + 1 \rightarrow \varphi_4) \land \]
\[(lc = N + 2 \rightarrow \varphi_5) \land \]

Let \( \alpha : lc >= 3 \rightarrow I = lc - 3 \land lc < 3 \rightarrow I = 0 \) and \( \kappa : pc = \pi. \)

The verification condition for the loop steady state is:
\( \alpha : \{I = lc - 3 \land pc = \pi \land \)
\[\alpha' : \{I' = lc' - 3 \land pc' = \pi' \land \)

11
\[ \varphi : \{ t_3 = b[lc - 1] \land t_2 = b[lc - 3] + 5 \land t_4 = b[lc - 2] \land i_1 = lc - 3 \land i_2 = lc \} \]

\[ \rho_t : \begin{cases} 
\forall k \in [0..N - 1] : (i_1 = k \land a'[i_1] = t_2 \lor a'[k] = a[k]) \land \\
\quad (i_2 = i_2 + 1 \land t_2 = b[i_2] \land t_2' = t_4 + 5 \land t_4' = t_3 \land \\
\quad lc' = lc + 1 \land pc = pc \land lc < N + 3 \land lc + 1 < N + 3 
\end{cases} \]

\[ \rho_s : \begin{cases} 
(\forall k \in [0..N - 1] : (k = I) \land a'[k] = b[k] + 5) \lor (a'[k] = a[k]) \land \\
I' = I + 1 \land \pi' = \pi \land I < N \land I + 1 < N \land \\
\varphi' : \{ t_3' = b[lc' - 1] \land t_2' = b[lc' - 3] + 5 \land t_4' = b[lc' - 2] \land i_1' = lc' - 3 \land i_2' = lc' \}
\]

This verification condition was verified by \texttt{step}[BBC98].

5.3 Validating the Verification Conditions

The verification conditions are of first order logic. Quantifiers are produced when handling memory variables, arrays in particular. Consider a verification condition transformed to a sequent \( \land \forall \alpha \rightarrow \forall \alpha' \). Universal quantifies may appear on the left side, as a result of the \( \alpha \) mapping (we assume that the verification conditions are without the existential quantifier, and we have both \( \alpha \) and \( \alpha' \) on the left side of the implication).

An array update that appears in the consequent part, such as: \( a[i] = \text{Exp} \) is replaced by the following equivalent function definition (skolemization, and the duality of arrays and functions): \( \text{if} \ (i = j) \ a'(i) = \text{Exp} \ \text{else} \ a'(i) = a(i) \) These manipulations produce verification conditions that are Pressburger formulae without quantifiers, and with uninterpreted functions.

References


tools and algorithms for the construction and analysis of systems*, volume 1384
of *tacas*, pages 151–166. Springer-Verlag, 98.


A SGI PRO64 produced 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 10: IA-64 assembly code

$r3, r2$ 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.