A Simple, Verified Validator for Software Pipelining

(verification pearl)

Jean-Baptiste Tristan
INRIA Paris-Rocquencourt
B.P. 105, 78153 Le Chesnay, France
jean-baptiste.tristan@inria.fr

Xavier Leroy
INRIA Paris-Rocquencourt
B.P. 105, 78153 Le Chesnay, France
xavier.leroy@inria.fr

Abstract

Software pipelining is a loop optimization that overlaps the execution of several iterations of a loop to expose more instruction-level parallelism. It can result in first-class performance characteristics, but at the cost of significant obfuscation of the code, making this optimization difficult to test and debug. In this paper, we present a translation validation algorithm that uses symbolic evaluation to detect semantics discrepancies between a loop and its pipelined version. Our algorithm can be implemented simply and efficiently, is provably sound, and appears to be complete with respect to most modulo scheduling algorithms. A conclusion of this case study is that it is possible and effective to use symbolic evaluation to reason about loop transformations.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification - Correctness proofs; D.3.4 [*Programming Languages*]: Processors - Optimization

General Terms Languages, Verification, Algorithms

Keywords Software pipelining, translation validation, symbolic evaluation, verified compilers

1. Introduction

> *There is one last technique in the arsenal of the software optimizer that may be used to make most machines run at tip top speed. It can also lead to severe code bloat and may make for almost unreadable code, so should be considered the last refuge of the truly desperate. However, its performance characteristics are in many cases unmatched by any other approach, so we cover it here. It is called software pipelining [. . .]*
>
> Apple Developer Connection[1]

Software pipelining is an advanced instruction scheduling optimization that exposes considerable instruction-level parallelism by overlapping the execution of several iterations of a loop. It produces smaller code and eliminates more pipeline stalls than merely unrolling the loop then performing acyclic scheduling. Software pipelining is implemented in many production compilers and described in several compiler textbooks [1, section 10.5] [2, chapter 20] [16, section 17.4]. In the words of the rather dramatic quote above, "truly desperate" programmers occasionally perform software pipelining by hand, especially on multimedia and signal processing kernels.

Starting in the 1980's, many clever algorithms were designed to produce efficient software pipelines and implement them either on stock hardware or by taking advantage of special features such as those of the IA64 architecture. In this paper, we are not concerned about the performance characteristics of these algorithms, but rather about their *semantic correctness*: does the generated, software-pipelined code compute the same results as the original code? As with all advanced compiler optimizations, and perhaps even more so here, mistakes happen in the design and implementation of software pipelining algorithms, causing incorrect code to be generated from correct source programs. The extensive code rearrangement performed by software pipelining makes visual inspection of the generated code ineffective; the additional boundary conditions introduced make exhaustive testing difficult. For instance, after describing a particularly thorny issue, Rau *et al.* [23] note that

> *The authors are indirectly aware of at least one computer manufacturer whose attempts to implement modulo scheduling, without having understood this issue, resulted in a compiler which generated incorrect code.*

Translation validation is a systematic technique to detect (at compile-time) semantic discrepancies introduced by buggy compiler passes or desperate programmers who optimize by hand, and to build confidence in the result of a compilation run or manual optimization session. In this approach, the programs before and after optimization are fed to a *validator* (a piece of software distinct from the optimizer), which tries to establish that the two programs are semantically equivalent; if it fails, compilation is aborted or continues with the unoptimized code, discarding the incorrect optimization. As invented by Pnueli *et al.* [20], translation validation proceeds by generation of verification conditions followed by model-checking or automated theorem proving [19, 29, 28, 3, 8]. An alternate approach, less general but less costly in validation time, relies on combinations of symbolic evaluation and static analysis [18, 24, 6, 26, 27]. The VCGen/theorem-proving approach was applied to software pipelining by Leviathan and Pnueli [14]. It was long believed that symbolic evaluation with static analyses was too weak to validate aggressive loop optimizations such as software pipelining.

In this paper, we present a novel translation validation algorithm for software pipelining, based on symbolic evaluation. This algorithm is surprisingly simple, proved to be sound (most of the

[1] `http://developer.apple.com/hardwaredrivers/ve/software_pipelining.html`

POPL'10, January 17–23, 2010, Madrid, Spain.

Original loop body $\mathcal{B}$:

```
x := load(float64, p);
y := x * c;
store(float64, p, y);
p := p + 8;
i := i + 1;
```

Prolog $\mathcal{P}$:

```
p1 := p;
p2 := p;
x1 := x;
x2 := x;
x1 := load(float64, p1);
p2 := p1 + 8;
x2 := load(float64, p2);
y := x1 * c;
i := i + 2;
```

Steady state $\mathcal{S}$:

```
store(float64, p1, y);
p1 := p2 + 8;
y := x2 * c;
x1 := load(float64, p1);
store(float64, p2, y);
p2 := p1 + 8;
y := x1 * c;
x2 := load(float64, p2);
i := i + 2;
```

Epilogue $\mathcal{E}$:

```
store(float64, p1, y);
y := x2 * c;
store(float64, p2, y);
x := x2;
p := p2;
```

Figure 1. An example of software pipelining

proof was mechanized using the Coq proof assistant), and informally argued to be complete with respect to a wide class of software pipelining optimizations.

The formal verification of a validator consists in mechanically proving that if it returns `true`, its two input programs do behave identically at run time. This is a worthwhile endeavor for two reasons. First, it brings additional confidence that the results of validation are trustworthy. Second, it provides an attractive alternative to formally verifying the soundness of the optimization algorithm itself: the validator is often smaller and easier to verify than the optimization it validates [26]. The validation algorithm presented in this paper grew out of the desire to add a software pipelining pass to the CompCert high-assurance C compiler [11]. Its formal verification in Coq is not entirely complete at the time of this writing, but appears within reach given the simplicity of the algorithm.

The remainder of this paper is organized as follows. Section 2 recalls the effect of software pipelining on the shape of loops. Section 3 outlines the basic principle of our validator. Section 4 defines the flavor of symbolic evaluation that it uses. Section 5 presents the validation algorithm. Its soundness is proved in section 6; its completeness is discussed in section 7. The experience gained on a prototype implementation is discussed in section 8. Section 9 discusses related work, and is followed by conclusions and perspectives in section 10.

2. Software pipelining

From the bird's eye, software pipelining is performed in three steps.

Step 1 Select one inner loop for pipelining. Like in most previous work, we restrict ourselves to simple counted loops of the form

```
i := 0;
while (i < N) { B }
```

We assume that the loop bound N is a loop-invariant variable, that the loop body $\mathcal{B}$ is a basic block (no conditional branches, no function calls), and that the loop index i is incremented exactly once in $\mathcal{B}$. The use of structured control above is a notational convenience: in reality, software pipelining is performed on a flow graph representation of control (CFG), and step 1 actually isolates a sub-graph of the CFG comprising a proper loop of the form above, as depicted in figure 2(a).

Step 2 Next, the *software pipeliner* is called. It takes as input the loop body $\mathcal{B}$ and the loop index i, and produces a 5-tuple $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$ as result. $\mathcal{P}$, $\mathcal{S}$ and $\mathcal{E}$ are sequences of non-branching instructions; μ and δ are positive integers.

- $\mathcal{S}$ is the new loop body, also called the *steady state* of the pipeline.
- $\mathcal{P}$ is the loop prolog: a sequence of instructions that fills the pipeline until it reaches the steady state.
- $\mathcal{E}$ is the loop epilog: a sequence of instructions that drains the pipeline, finishing the computations that are still in progress at the end of the steady state.
- μ is the minimum number of iterations that must be performed to be able to use the pipelined loop.
- δ is the amount of unrolling that has been performed on the steady state. In other words, one iteration of $\mathcal{S}$ corresponds to δ iterations of the original loop $\mathcal{B}$.

The prolog $\mathcal{P}$ is assumed to increment the loop index i by μ; the steady state $\mathcal{S}$, by δ; and the epilogue $\mathcal{E}$, not at all.

The software pipelining algorithms that we have in mind here are combinations of *modulo scheduling* [22, 7, 15, 23, 4] followed by *modulo variable expansion* [10]. However, the presentation above seems general enough to accommodate other scheduling algorithms.

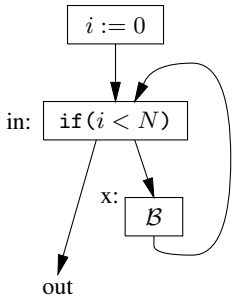
Figure 1 illustrates one run of a software pipeliner. The schedule obtained by modulo scheduling performs, at each iteration, the i^{th} `store` and increment of `p`, the $(i+1)^{\text{th}}$ multiplication by `c`, and the $(i+2)^{\text{th}}$ `load`. To avoid read-after-write hazards, modulo variable expansion was performed, unrolling the schedule by a factor of 2, and replacing variables `p` and `x` by two variables each, `p1`/`p2` and `x1`/`x2`, which are defined in an alternating manner. These pairs of variables are initialized from `p` and `x` in the prolog; the epilogue sets `p` and `x` back to their final values. The unrolling factor δ is therefore 2. Likewise, the minimum number of iterations is $\mu = 2$.

Step 3 Finally, the original loop is replaced by the following pseudo-code:

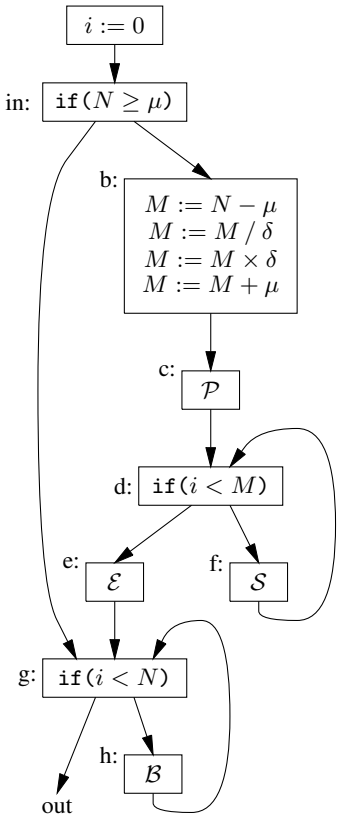
```
i := 0;
if (N ≥ μ) {
  M := ((N − μ)/δ) × δ + μ;
  P
  while (i < M) { S }
  E
}
while (i < N) { B }
```

In CFG terms, this amounts to replacing the sub-graph shown at the top of figure 2 by the one shown at the bottom.

The effect of the code after pipelining is as follows. If the number of iterations N is less than μ, a copy of the original loop is executed. Otherwise, we execute the prolog $\mathcal{P}$, then iterate the steady state $\mathcal{S}$, then execute the epilogue $\mathcal{E}$, and finally fall through the copy of the original loop. Since $\mathcal{P}$ and $\mathcal{S}$ morally correspond to μ and δ iterations of the original loop, respectively, $\mathcal{S}$ is iterated n times where n is the largest integer such that $\mu + n\delta \leq N$, namely $n = (N - \mu)/\delta$. In terms of the loop index `i`, this corresponds to the upper limit M shown in the pseudo-code above. The copy of the original loop executes the remaining $N - M$ iterations.



(a) Original sub-CFG



(b) Replacement sub-CFG after software pipelining

Figure 2. The effect of software pipelining on a control-flow graph

Steps 1, 2 and 3 can of course be repeated once per inner loop eligible for pipelining. In the remainder of this paper, we focus on the transformation of a single loop.

3. Validation a posteriori

In a pure translation validation approach, the validator would receive as inputs the code before and after software pipelining and would be responsible for establishing the correctness of all three steps of pipelining. We found it easier to concentrate translation validation on step 2 only (the construction of the components of the pipelined loop) and revert to traditional compiler verification for steps 1 and 3 (the assembling of these components to form the transformed code). In other words, the validator we set out to build takes as arguments the input $(i, N, \mathcal{B})$ and output $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$ of the software pipeliner (step 2). It is responsible for establishing conditions sufficient to prove semantic preservation between the original and transformed codes (top and bottom parts of figure 2).

What are these sufficient conditions and how can a validator establish them? To progress towards an answer, think of complete unrollings of the original and pipelined loops. We see that the runtime behavior of the original loop is equivalent to that of $\mathcal{B}^N$, that is, N copies of $\mathcal{B}$ executed in sequence. Likewise, the generated code behaves either like $\mathcal{B}^N$ if $N < \mu$, or like

$$\mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)}$$

if $N \geq \mu$, where

$$\kappa(N) \stackrel{\text{def}}{=} (N - \mu)/\delta$$
$$\rho(N) \stackrel{\text{def}}{=} N - \mu - \delta \times \kappa(N)$$

The verification problem therefore reduces to establishing that, for all N, the two basic blocks

$$X_N \stackrel{\text{def}}{=} \mathcal{B}^N \quad \text{and} \quad Y_N \stackrel{\text{def}}{=} \mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)}$$

are semantically equivalent: if both blocks are executed from the same initial state (same memory state, same values for variables), they terminate with the same final state (up to the values of temporary variables introduced during scheduling and unused later).

For a given value of N, *symbolic evaluation* provides a simple, effective way to ensure this semantic equivalence between X_N and Y_N. In a nutshell, symbolic evaluation is a form of denotational semantics for basic blocks where the values of variables $x, y, \ldots$ at the end of the block are determined as symbolic expressions of their values $x_0, y_0, \ldots$ at the beginning of the block. For example, the symbolic evaluation of the block $B \stackrel{\text{def}}{=} y := x+1;\ x := y \times 2$ is

$$\alpha(B) = \left\{ \begin{array}{lcl} x & \mapsto & (x_0 + 1) \times 2 \\ y & \mapsto & x_0 + 1 \\ v & \mapsto & v_0 \text{ for all } v \notin \{x, y\} \end{array} \right.$$

(This is a simplified presentation of symbolic evaluation; section 4 explains how to extend it to handle memory accesses, potential run-time errors, and the fresh variables introduced by modulo variable expansion during pipelining.)

The fundamental property of symbolic evaluation is that if two blocks have identical symbolic evaluations, whenever they are executed in the same but arbitrary initial state, they terminate in the same final state. Moreover, symbolic evaluation is insensitive to reordering of independent instructions, making it highly suitable to reason over scheduling optimizations [26]. Therefore, for any given N, our validator can check that $\alpha(X_N) = \alpha(Y_N)$; this suffices to guarantee that X_N and Y_N are semantically equivalent.

If N were a compile-time constant, this would provide us with a validation algorithm. However, N is in general a run-time quantity, and statically checking $\alpha(X_N) = \alpha(Y_N)$ for all N is not decidable. The key discovery of this paper is that this undecidable property $\forall N,\ \alpha(X_N) = \alpha(Y_N)$ is implied by (and, in practice, equivalent to) the following two decidable conditions:

$$\alpha(\mathcal{E}; \mathcal{B}^\delta) = \alpha(\mathcal{S}; \mathcal{E}) \quad (1)$$
$$\alpha(\mathcal{B}^\mu) = \alpha(\mathcal{P}; \mathcal{E}) \quad (2)$$

The programmer's intuition behind condition 1 is the following: in a correct software pipeline, if enough iterations remain to be performed, it should always be possible to choose between 1) execute the steady state $\mathcal{S}$ one more time, then leave the pipelined loop and execute $\mathcal{E}$; or 2) leave the pipelined loop immediately by executing $\mathcal{E}$, then run δ iterations of the original loop body $\mathcal{B}$. (See figure 3.) Condition 2 is even easier to justify: if the number

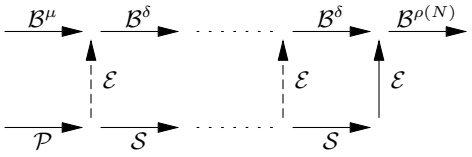


Figure 3. A view of an execution of the loop, before (top) and after (bottom) pipelining. The dashed vertical arrows provide intuitions for equations 1 and 2.

of executions is exactly μ, the original code performs $\mathcal{B}^{\mu}$ and the pipelined code performs $\mathcal{P};\mathcal{E}$.

The translation validation algorithm that we define in section 5 is based on checking minor variations of conditions 1 and 2, along with some additional conditions on the evolutions of variable i (also checked using symbolic evaluation). Before presenting this algorithm, proving that it is sound, and arguing that it is complete in practice, we first need to define symbolic evaluation more precisely and study its algebraic structure.

4. Symbolic evaluation with observables

In this section, we formally define the form of symbolic evaluation used in our validator. Similar forms of symbolic evaluation are at the core of several other translation validators, such as those of Necula [18], Rival [24], and Tristan and Leroy [26].

4.1 The intermediate language

We perform symbolic evaluation over the following simple language of basic blocks:

Non-branching instructions:

I	$::= r := r'$	variable-variable move
	$\mid r := \texttt{op}(op, \vec{r})$	arithmetic operation
	$\mid r := \texttt{load}(\kappa, r_{addr})$	memory load
	$\mid \texttt{store}(\kappa, r_{addr}, r_{val})$	memory store

Basic blocks:

B	$::= I_1; \ldots; I_n$	instruction sequences

Here, r ranges over variable names (a.k.a. pseudo-registers or temporaries); op ranges over arithmetic operators (such as "load integer constant n" or "add integer immediate n" or "float multiply"); and κ stands for a memory quantity (such as "8-bit signed integer" or "64-bit float").

4.2 Symbolic states

The symbolic evaluation of a block produces a pair (φ, σ) of a mapping φ from resources to terms, capturing the relationship between the initial and final values of resources, and a set of terms σ, recording the computations performed by the block.

Resources:

$$\rho ::= r \mid \texttt{Mem}$$

Terms:

t	$::= \rho_0$	initial value of ρ
	$\mid \texttt{Op}(op, \vec{t})$	result of an operation
	$\mid \texttt{Load}(\kappa, t_{addr}, t_m)$	result of a load
	$\mid \texttt{Store}(\kappa, t_{addr}, t_{val}, t_m)$	effect of a store

Resource maps:

φ	$::= \rho \mapsto t$	finite map

Computations performed:

σ	$::= \{t_1; \ldots; t_n\}$	finite set

Symbolic states:

$$A ::= (\varphi, \sigma)$$

Resource maps symbolically track the values of variables and the memory state, represented like a ghost variable $\texttt{Mem}$. A resource ρ that is not in the domain of a map φ is considered mapped to ρ_0. Memory states are represented by terms t_m built out of $\texttt{Mem}_0$ and $\texttt{Store}(\ldots)$ constructors. For example, the following block

$$\texttt{store}(\kappa, x, y);\ \ z := \texttt{load}(\kappa, w)$$

symbolically evaluates to the following resource map:

$$\begin{aligned} \texttt{Mem} &\mapsto \texttt{Store}(\kappa, x_0, y_0, \texttt{Mem}_0) \\ z &\mapsto \texttt{Load}(\kappa, w_0, \texttt{Store}(\kappa, x_0, y_0, \texttt{Mem}_0)) \end{aligned}$$

Resource maps describe the final state after execution of a basic block, but do not capture all intermediate operations performed. This is insufficient if some intermediate operations can fail at run-time (*e.g.*integer divisions by zero or loads from a null pointer). Consider for example the two blocks

$$y := x \quad \text{and} \quad \begin{array}{l} y := \texttt{load}(\texttt{int32}, x); \\ y := x \end{array}$$

They have the same resource map, namely $y \mapsto x_0$, yet the rightmost block right can cause a run-time error if the pointer x is null, while the leftmost block never fails. Therefore, resource maps do not suffice to check that two blocks have the same semantics; they must be complemented by sets σ of symbolic terms, recording all the computations that are performed by the blocks, even if these computations do not contribute directly to the final state. Continuing the example above, the leftmost block has $\sigma = \emptyset$ (a move is not considered as a computation, since it cannot fail) while the rightmost block has $\sigma' = \{\texttt{Load}(\texttt{int32}, x_0, \texttt{Mem}_0)\}$. Since these two sets differ, symbolic evaluation reveals that the two blocks above are not semantically equivalent with respect to run-time errors.

4.3 Composition

A resource map φ can be viewed as a parallel substitution of symbolic terms for resources: the action of a map on a term t is the term $\varphi(t)$ obtained by replacing all occurrences of ρ_0 in t by the term associated with ρ in φ. Therefore, the reverse composition $\varphi_1;\varphi_2$ of two maps is, classically,

$$\varphi_1;\varphi_2 \stackrel{\text{def}}{=} \{\rho \mapsto \varphi_1(\varphi_2(\rho)) \mid \rho \in \mathrm{Dom}(\varphi_1) \cup \mathrm{Dom}(\varphi_2)\} \quad (3)$$

The reverse composition operator extends naturally to abstract states:

$$(\varphi_1, \sigma_1);(\varphi_2, \sigma_2) \stackrel{\text{def}}{=} (\varphi_1;\varphi_2,\ \sigma_1 \cup \{\varphi_1(t) \mid t \in \sigma_2\}) \quad (4)$$

Composition is associative and admits the identity abstract state $\varepsilon \stackrel{\text{def}}{=} (\emptyset, \emptyset)$ as neutral element.

4.4 Performing symbolic evaluation

The symbolic evaluation $\alpha(I)$ of a single instruction I is the abstract state defined by

$$\begin{aligned} \alpha(r := r') &= (r \mapsto r'_0,\ \emptyset) \\ \alpha(r := \texttt{op}(op, \vec{r})) &= (r \mapsto t,\ \{t\}) \\ &\quad \text{where } t = \texttt{Op}(op, \vec{r_0}) \\ \alpha(r := \texttt{load}(\kappa, r')) &= (r \mapsto t,\ \{t\}) \\ &\quad \text{where } t = \texttt{Load}(\kappa, r'_0, \texttt{Mem}_0) \\ \alpha(\texttt{store}(\kappa, r, r')) &= (\texttt{Mem} \mapsto t_m,\ \{t_m\}) \\ &\quad \text{where } t_m = \texttt{Store}(\kappa, r_0, r'_0, \texttt{Mem}_0) \end{aligned}$$

Symbolic evaluation then extends to basic blocks, by composing the evaluations of individual instructions:

$$\alpha(I_1; \ldots; I_n) \stackrel{\text{def}}{=} \alpha(I_1); \ldots; \alpha(I_n) \tag{5}$$

By associativity, it follows that symbolic evaluation distributes over concatenation of basic blocks:

$$\alpha(B_1; B_2) = \alpha(B_1); \alpha(B_2) \tag{6}$$

4.5 Comparing symbolic states

We use two notions of equivalence between symbolic states. The first one, written $\sim$, is defined as strict syntactic equality:[2]

$$(\varphi, \sigma) \sim (\varphi', \sigma') \text{ if and only if } \forall \rho,\ \varphi(\rho) = \varphi'(\rho) \text{ and } \sigma = \sigma' \tag{7}$$

The relation $\sim$ is therefore an equivalence relation, and is compatible with composition:

$$A_1 \sim A_2 \Longrightarrow A_1; A \sim A_2; A \tag{8}$$

$$A_1 \sim A_2 \Longrightarrow A; A_1 \sim A; A_2 \tag{9}$$

We need a coarser equivalence between symbolic states to account for the differences in variable usage between the code before and after software pipelining. Recall that pipelining may perform modulo variable expansion to avoid read-after-write hazards. This introduces fresh temporary variables in the pipelined code. These fresh variables show up in symbolic evaluations and prevent strict equivalence $\sim$ from holding. Consider:

$$x := \mathtt{op}(op, x) \quad \text{and} \quad \begin{array}{l} x' := \mathtt{op}(op, x) \\ x := x' \end{array}$$

These two blocks have different symbolic evaluations (in the sense of the $\sim$ equivalence), yet should be considered as semantically equivalent if the temporary x' is unused later.

Let θ be a finite set of variables: the *observable* variables. (In our validation algorithm, we take θ to be the set of all variable mentioned in the original code before pipelining; the fresh temporaries introduced by modulo variable expansion are therefore not in θ.) Equivalence between symbolic states up to the observables θ is written $\approx_\theta$ and defined as

$$(\varphi, \sigma) \approx_\theta (\varphi', \sigma') \text{ if and only if}$$
$$\forall \rho \in \theta \cup \{\mathtt{Mem}\},\ \varphi(\rho) = \varphi'(\rho) \text{ and } \sigma = \sigma' \tag{10}$$

The relation $\approx_\theta$ is an equivalence relation, coarser than $\sim$, and compatible with composition on the right:

$$A_1 \sim A_2 \Longrightarrow A_1 \approx_\theta A_2 \tag{11}$$

$$A_1 \approx_\theta A_2 \Longrightarrow A; A_1 \approx_\theta A; A_2 \tag{12}$$

However, it is not, in general, compatible with composition on the left. Consider an abstract state A that maps a variable $x \in \theta$ to a term t containing an occurrence of $y \notin \theta$. Further assume $A_1 \approx_\theta A_2$. The composition $A_1; A$ maps x to $A_1(t)$; likewise, $A_2; A$ maps x to $A_2(t)$. Since $y \notin \theta$, we can have $A_1(y) \neq A_2(y)$ and therefore $A_1(t) \neq A_2(t)$ without violating the hypothesis $A_1 \approx_\theta A_2$. Hence, $A_1; A \approx_\theta A_2; A$ does not hold in general.

To address this issue, we restrict ourselves to symbolic states A that are *contained* in the set θ of observables. We say that a symbolic term t is contained in θ, and write $t \sqsubseteq \theta$, if all variables x appearing in t belong to θ. We extend this notion to symbolic states as follows:

$$(\varphi, \sigma) \sqsubseteq \theta \text{ if and only if}$$
$$\forall \rho \in \theta \cup \{\mathtt{Mem}\},\ \varphi(\rho) \sqsubseteq \theta \text{ and } \forall t \in \sigma,\ t \sqsubseteq \theta \tag{13}$$

It is easy to see that $\approx_\theta$ is compatible with composition on the left if the right symbolic state is contained in θ:

$$A_1 \approx_\theta A_2 \ \wedge\ A \sqsubseteq \theta \Longrightarrow A_1; A \approx_\theta A_2; A \tag{14}$$

The containment relation enjoys additional useful properties:

$$A_1 \sqsubseteq \theta \ \wedge\ A_1 \approx_\theta A_2 \Longrightarrow A_2 \sqsubseteq \theta \tag{15}$$

$$A_1 \sqsubseteq \theta \ \wedge\ A_2 \sqsubseteq \theta \Longrightarrow (A_1; A_2) \sqsubseteq \theta \tag{16}$$

Semantic interpretation of arithmetic and memory operations:

$$\begin{array}{rcl} \overline{op} & : & \mathtt{list\ val} \to \mathtt{option\ val} \\ \overline{\mathtt{load}} & : & \mathtt{quantity} \times \mathtt{val} \times \mathtt{mem} \to \mathtt{option\ val} \\ \overline{\mathtt{store}} & : & \mathtt{quantity} \times \mathtt{val} \times \mathtt{val} \times \mathtt{mem} \to \mathtt{option\ mem} \end{array}$$

Transition semantics for instructions:

$$r := r' : (R, M) \to (R[r \leftarrow R(r')], M)$$

$$\frac{\overline{op}(R(\vec{r})) = \lfloor v \rfloor}{r := \mathtt{op}(op, \vec{r}) : (R, M) \to (R[r \leftarrow v], M)}$$

$$\frac{\overline{\mathtt{load}}(\kappa, R(r_{addr}), M) = \lfloor v \rfloor}{r := \mathtt{load}(\kappa, r_{addr}) : (R, M) \to (R[r \leftarrow v], M)}$$

$$\frac{\overline{\mathtt{store}}(\kappa, R(r_{addr}), R(r_{val}), M) = \lfloor M' \rfloor}{\mathtt{store}(\kappa, r_{addr}, r_{val}) : (R, M) \to (R, M')}$$

Transition semantics for basic blocks:

$$\varepsilon : S \stackrel{*}{\to} S \qquad \frac{I : S \to S' \qquad B : S' \stackrel{*}{\to} S''}{I.B : S \stackrel{*}{\to} S''}$$

Figure 4. Operational semantics for basic blocks

4.6 Semantic soundness

The fundamental property of symbolic evaluation is that if $\alpha(B_1) \approx_\theta \alpha(B_2)$, the two blocks B_1 and B_2 have the same run-time behavior, in a sense that we now make precise.

We equip our language of basic blocks with the straightforward operational semantics shown in figure 4. The behavior of arithmetic and memory operations is axiomatized as functions $\overline{op}$, $\overline{\mathtt{load}}$ and $\overline{\mathtt{store}}$ that return "option" types to model run-time failures: the result $\emptyset$ (pronounced "none") denotes failure; the result $\lfloor x \rfloor$ (pronounced "some x") denotes success. In our intended application, these interpretation functions are those of the RTL intermediate language of the CompCert compiler [12, section 6.1] and of its memory model [13].

The semantics for a basic block B is, then, given as the relation $B : S \stackrel{*}{\to} S'$, sometimes also written $S \stackrel{B}{\to} S'$, where S is the initial state and S' the final state. States are pairs (R, M) of a variable state R, mapping variables to values, and a memory state M.

We say that two execution states $S = (R, M)$ and $S' = (R', M')$ are equivalent up to the observables θ, and write $S \cong_\theta S'$, if $M = M'$ and $R(r) = R'(r)$ for all $r \in \theta$.

THEOREM 1 (Soundness of symbolic evaluation). *Let B_1 and B_2 be two basic blocks. Assume that $\alpha(B_1) \approx_\theta \alpha(B_2)$ and $\alpha(B_2) \sqsubseteq$*

[2] We could relax this definition in two ways. One is to compare symbolic terms up to a decidable equational theory capturing algebraic identities such as $x \times 2 = x + x$. This enables the validation of *e.g.* instruction strength reduction [18, 24]. Another relaxation is to require $s' \subseteq s$ instead of $s = s'$, enabling the transformed basic block to perform fewer computations than the original block (*e.g.* dead code elimination). Software pipelining being a purely lexical transformation that changes only the placement of computations but not their nature nor their number, these two relaxations are not essential and we stick with strict syntactic equality for the time being.

θ. *If* $B_1 : S \xrightarrow{*} S'$ *and* $S \cong_\theta T$, *there exists* T' *such that* $B_2 : T \xrightarrow{*} T'$ *and* $S' \cong_\theta T'$.

We omit the proof, which is similar to that of Lemma 3 in [26].

5. The validation algorithm

We can now piece together the intuitions from section 3 and the definitions from section 4 to obtain the following validation algorithm:

$$
\begin{array}{ll}
\texttt{validate}\ (i, N, \mathcal{B})\ (\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)\ \theta = & \\
\quad \alpha(\mathcal{B}^\mu) \approx_\theta \alpha(\mathcal{P}; \mathcal{E}) & \text{(A)} \\
\wedge\ \alpha(\mathcal{E}; \mathcal{B}^\delta) \approx_\theta \alpha(\mathcal{S}; \mathcal{E}) & \text{(B)} \\
\wedge\ \alpha(\mathcal{B}) \sqsubseteq \theta & \text{(C)} \\
\wedge\ \alpha(\mathcal{B})(i) = \texttt{Op}(\texttt{addi}(1), i_0) & \text{(D)} \\
\wedge\ \alpha(\mathcal{P})(i) = \alpha(\mathcal{B}^\mu)(i) & \text{(E)} \\
\wedge\ \alpha(\mathcal{S})(i) = \alpha(\mathcal{B}^\delta)(i) & \text{(F)} \\
\wedge\ \alpha(\mathcal{E})(i) = i_0 & \text{(G)} \\
\wedge\ \alpha(\mathcal{B})(N) = \alpha(\mathcal{P})(N) = \alpha(\mathcal{S})(N) = \alpha(\mathcal{E})(N) = N_0 & \text{(H)}
\end{array}
$$

The inputs of the validator are the components of the loops before $(i, N, \mathcal{B})$ and after $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$ pipelining, as well as the set θ of observables. Checks A and B correspond to equations (1) and (2) of section 3, properly reformulated in terms of equivalence up to observables. Check C makes sure that the set θ of observables includes at least the variables appearing in the symbolic evaluation of the original loop body $\mathcal{B}$. Checks D to G ensure that the loop index i is incremented the way we expect it to be: by one in $\mathcal{B}$, by μ in $\mathcal{P}$, by δ in $\mathcal{S}$, and not at all in $\mathcal{E}$. (We write $\alpha(B)(i)$ to refer to the symbolic term associated to i by the resource map part of the symbolic state $\alpha(B)$.) Finally, check H makes sure that the loop limit N is invariant in both loops.

The algorithm above has low computational complexity. Using a hash-consed representation of symbolic terms, the symbolic evaluation $\alpha(B)$ of a block B containing n instructions can be computed in time $O(n \log n)$: for each of the n instructions, the algorithm performs one update on the resource map, one insertion in the constraint set, and one hash-consing operation, all of which can be done in logarithmic time. (The resource map and the constraint set have sizes at most n.) Likewise, the equivalence and containment relations ($\approx_\theta$ and $\sqsubseteq$) can be decided in time $O(n \log n)$. Finally, note that this $O(n \log n)$ complexity holds even if the algorithm is implemented within the Coq specification language, using persistent, purely functional data structures.

The running time of the validation algorithm is therefore $O(n \log n)$ where $n = \max(\mu|\mathcal{B}|, \delta|\mathcal{B}|, |\mathcal{P}|, |\mathcal{S}|, |\mathcal{E}|)$ is proportional to the size of the loop after pipelining. Theoretically, software pipelining can increase the size of the loop quadratically. In practice, we expect the external implementation of software pipelining to avoid this blowup and produce code whose size is linear in that of the original loop. In this case, the validator runs in time $O(n \log n)$ where n is the size of the original loop.

6. Soundness proof

In this section, we prove the soudness of the validator: if it returns "true", the software pipelined loop executes exactly like the original loop. The proof proceeds in two steps: first, we show that the basic blocks X_N and Y_N obtained by unrolling the two loops have the same symbolic evaluation and therefore execute identically; then, we prove that the concrete execution of the two loops in the CFG match.

6.1 Soundness with respect to the unrolled loops

We first show that the finitary conditions checked by the validator imply the infinitary equivalences between the symbolic evaluations of X_N and Y_N sketched in section 3.

LEMMA 2. *If* `validate` $(i, N, \mathcal{B})$ $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$ θ *returns* `true`, *then, for all* n,

$$\alpha(\mathcal{B}^{\mu+n\delta}) \approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E}) \tag{17}$$

$$\alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E}) \sqsubseteq \theta \tag{18}$$

$$\alpha(\mathcal{B}^{\mu+n\delta})(i) = \alpha(\mathcal{P}; \mathcal{S}^n)(i) = \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E})(i) \tag{19}$$

Proof: By hypothesis, we know that the properties A to H checked by the validator hold. First note that

$$\forall n,\ \ \alpha(\mathcal{B}^n) \sqsubseteq \theta \tag{20}$$

as a consequence of check C and property (16).

Conclusion (17) is proved by induction on n. The base case $n = 0$ reduces to $\alpha(\mathcal{B}^\mu) \approx_\theta \alpha(\mathcal{P}; \mathcal{E})$, which is ensured by check A. For the inductive case, assume $\alpha(\mathcal{B}^{\mu+n\delta}) \approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E})$.

$$
\begin{aligned}
\alpha(\mathcal{B}^{\mu+(n+1)\delta}) &\approx_\theta \alpha(\mathcal{B}^{\mu+n\delta}); \alpha(\mathcal{B}^\delta) \\
&\quad \text{(by distributivity (6))} \\
&\approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E}); \alpha(\mathcal{B}^\delta) \\
&\quad \text{(by left compatibility (14) and ind. hyp.)} \\
&\approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n); \alpha(\mathcal{E}; \mathcal{B}^\delta) \\
&\quad \text{(by distributivity (6))} \\
&\approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n); \alpha(\mathcal{S}; \mathcal{E}) \\
&\quad \text{(by check B, right compatibility (12), and (20))} \\
&\approx_\theta \alpha(\mathcal{P}; \mathcal{S}^{n+1}; \mathcal{E}) \\
&\quad \text{(by distributivity (6))}
\end{aligned}
$$

Conclusion (18) follows from (17), (20), and property (15). Conclusion (19) follows from checks E, F and G by induction on n. □

THEOREM 3. *Assume that* `validate` $(i, N, \mathcal{B})$ $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$ θ *returns* `true`. *Consider an execution* $\mathcal{B}^N : S \xrightarrow{*} S'$ *of the unrolled initial loop, with* $N \geq \mu$. *Assume* $S \cong_\theta T$. *Then, there exists a concrete state* T' *such that*

$$\mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)} : T \xrightarrow{*} T' \tag{21}$$

$$S' \cong_\theta T' \tag{22}$$

Moreover, execution (21) decomposes as follows:

$$T \xrightarrow{\mathcal{P}} T_0 \underbrace{\xrightarrow{\mathcal{S}} \cdots \xrightarrow{\mathcal{S}}}_{\kappa(N)\ \text{times}} T_{\kappa(N)} \xrightarrow{\mathcal{E}} T'_0 \underbrace{\xrightarrow{\mathcal{B}} \cdots \xrightarrow{\mathcal{B}}}_{\rho(N)\ \text{times}} T'_{\rho(N)} = T' \tag{23}$$

and the values of the loop index i *in the various intermediate states satisfy*

$$T_j(i) = S(i) + \mu + \delta j \pmod{2^{32}} \tag{24}$$

$$T'_j(i) = S(i) + \mu + \delta \times \kappa(N) + j \pmod{2^{32}} \tag{25}$$

Proof: As a corollary of Lemma 2, properties (17) and (18), we obtain

$$\alpha(\mathcal{B}^N) \approx_\theta \alpha(\mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)})$$

$$\alpha(\mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)}) \sqsubseteq \theta$$

since $N = \mu + \delta \times \kappa(N) + \rho(N)$. The existence of T' then follows from Theorem 1.

It is obvious that the concrete execution of the pipelined code can be decomposed as shown in (23). Check D in the validator

CFG instructions:

$$\begin{array}{lll} I_g ::= (I, \ell) & & \text{non-branching instr.} \\ \quad \mid (\texttt{if}(r < r'), \ell_t, \ell_f) & & \text{conditional branch} \end{array}$$

CFG:

$$g \ ::= \ell \mapsto I_g \qquad \text{finite map}$$

Transition semantics:

$$\frac{g(\ell) = (I, \ell') \qquad I : (R, M) \to (R', M') \text{ (as in figure 4)}}{g : (\ell, R, M) \to (\ell', R', M')}$$

$$\frac{g(\ell) = (\texttt{if}(r < r'), \ell_t, \ell_f) \qquad \ell' = \begin{cases} \ell_t & \text{if } R(r) < R(r') \\ \ell_f & \text{if } R(r) \geq R(r') \end{cases}}{g : (\ell, R, M) \to (\ell', R, M)}$$

Figure 5. A simple language of control-flow graphs

guarantees that $S_{j+1}(i) = S_j(i) + 1 \pmod{2^{32}}$, which implies $S_j(i) = S(i) + j \bmod 2^{32}$. Combined with conclusion (19) of Lemma 2, this implies equation (24):

$$T_j(i) = S_{\mu+\delta j}(i) = S(i) + \mu + \delta j \pmod{2^{32}}$$

as well as

$$T'_0(i) = S_{\mu+\delta\times\kappa(N)}(i) = S(i) + \mu + \delta \times \kappa(N) \pmod{2^{32}}$$

Check D entails that $T'_j(i) = T'_0(i) + j \bmod 2^{32}$. Equation (25) therefore follows. □

6.2 Soundness with respect to CFG loops

The second and last part of the soundness proof extends the semantic preservation results of theorem 3 from the unrolled loops in a basic-block representation to the actual loops in a CFG (control-flow graph) representation, namely the loops before and after pipelining depicted in figure 2. This extension is intuitively obvious but surprisingly tedious to formalize in full details: indeed, this is the only result in this paper that we have not yet mechanized in Coq.

Our final objective is to perform instruction scheduling on the RTL intermediate language from the CompCert compiler [12, section 6.1]. To facilitate exposition, we consider instead the simpler language of control-flow graphs defined in figure 5. The CFG g is represented as partial map from labels ℓ to instructions I_g. An instruction can be either a non-branching instruction I or a conditional branch $\texttt{if}(r < r')$, complemented by one or two labels denoting the successors of the instruction. The operational semantics of this language is given by a transition relation $g : (\ell, R, M) \to (\ell', R', M')$ between states comprising a program point ℓ, a register state R and a memory state M.

The following trivial lemma connects the semantics of basic blocks (figure 4) with the semantics of CFGs (figure 5). We say that a CFG g contains the basic block B between points ℓ and ℓ', and write $g, B : \ell \rightsquigarrow \ell'$, if there exists a path in g from ℓ to ℓ' that "spells out" the instructions $I_1, \ldots, I_n$ of B:

$$g(\ell) = (I_1, \ell_1) \wedge g(\ell_1) = (I_2, \ell_2) \wedge \cdots \wedge g(\ell_n) = (I_n, \ell')$$

LEMMA 4. *Assume $g, B : \ell \rightsquigarrow \ell'$. Then, $g : (\ell, S) \xrightarrow{*} (\ell', S')$ if and only if $B : S \xrightarrow{*} S'$.*

In the remainder of this section, we consider two control-flow graphs: g for the original loop before pipelining, depicted in part (a) of figure 2, and g' for the loop after pipelining, depicted in part (b) of figure 2. (Note that figure 2 is not a specific example: it describes the general shape of CFGs accepted and produced by all software pipelining optimizations we consider in this paper.) We now use lemma 4 to relate a CFG execution of the original loop g with the basic-block execution of its unrolling.

LEMMA 5. *Assume $\alpha(\mathcal{B})(i) = \texttt{Op}(\texttt{addi}(1), i_0)$ and $\alpha(\mathcal{B})(N) = N_0$. Consider an execution $g : (\texttt{in}, S) \xrightarrow{*} (\texttt{out}, S')$ from point* `in` *to point* `out` *in the original CFG, with $S(i) = 0$. Then, $\mathcal{B}^N : S \xrightarrow{*} S'$ where N is the value of variable N in state S.*

Proof: The CFG execution can be decomposed as follows:

$$(\texttt{in}, S_0) \to (\texttt{x}, S_0) \xrightarrow{*} (\texttt{in}, S_1) \to \cdots \xrightarrow{*} (\texttt{in}, S_n) \to (\texttt{out}, S_n)$$

with $S = S_0$ and $S' = S_n$ and $S_n(i) \geq S_n(N)$ and $S_j(i) < S_j(N)$ for all $j < n$. By construction of $\mathcal{B}$, we have $g, \mathcal{B} : \texttt{x} \rightsquigarrow \texttt{in}$. Therefore, by Lemma 4, $\mathcal{B} : S_j \xrightarrow{*} S_{j+1}$ for $j = 0, \ldots, n-1$. It follows that $\mathcal{B}^n : S \xrightarrow{*} S'$. Moreover, at each loop iteration, variable N is unchanged and variable i is incremented by 1. It follows that the number n of iterations is equal to the value N of variable N in initial state S. This is the expected result. □

Symmetrically, we can reconstruct a CFG execution of the pipelined loop g' from the basic-block execution of its unrolling.

LEMMA 6. *Let T_{init} be an initial state where variable i has value 0 and variable N has value N. Let T be T_{init} where variable M is set to $\mu + \delta \times \kappa(N)$. Assume that properties (23), (24) and (25) hold, as well as check H. We can, then, construct an execution $g' : (\texttt{in}, T_{init}) \xrightarrow{*} (\texttt{out}, T')$ from point* `in` *to point* `out` *in the CFG g' after pipelining.*

Proof: By construction of g', we have $g', \mathcal{P} : \texttt{c} \rightsquigarrow \texttt{d}$ and $g', \mathcal{S} : \texttt{f} \rightsquigarrow \texttt{d}$ and $g', \mathcal{E} : \texttt{e} \rightsquigarrow \texttt{g}$ and $g', \mathcal{B} : \texttt{h} \rightsquigarrow \texttt{g}$. The result is obvious if $N < \mu$. Otherwise, applying Lemma 4 to the basic-block executions given by property (23), we obtain the following CFG executions:

$$\begin{array}{l} g' : (\texttt{in}, T_{init}) \xrightarrow{*} (\texttt{c}, T) \\ g' : (\texttt{c}, T) \xrightarrow{*} (\texttt{d}, T_0) \\ g' : (\texttt{f}, T_j) \xrightarrow{*} (\texttt{d}, T_{j+1}) \text{ for } j = 0, \ldots, \kappa(N) - 1 \\ g' : (\texttt{e}, T_{\kappa(N)}) \xrightarrow{*} (\texttt{g}, T'_0) \\ g' : (\texttt{h}, T'_j) \xrightarrow{*} (\texttt{g}, T'_{j+1}) \text{ for } j = 0, \ldots, \rho(N) - 1 \end{array}$$

The variables N and M are invariant between points `c` and `out`: N because of check H, M because it is a fresh variable. Using properties (24) and (25), we see that the condition at point `d` is true in states $T_0, \ldots, T_{\kappa(N)-1}$ and false in state $T_{\kappa(N)}$. Likewise, the condition at point `g` is true in states $T'_0, \ldots, T'_{\rho(N)-1}$ and false in state $T'_{\rho(N)}$. The expected result follows. □

Piecing everything together, we obtain the desired semantic preservation result.

THEOREM 7. *Assume that* `validate` $(i, N, \mathcal{B})$ $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$ θ *returns* `true`. *Let g and g' be the CFG before and after pipelining, respectively. Consider an execution $g : (\texttt{in}, S) \xrightarrow{*} (\texttt{out}, S')$ with $S(i) = 0$. Then, for all states T such that $S \cong_\theta T$, there exists a state T' such that $g' : (\texttt{in}, T) \xrightarrow{*} (\texttt{out}, T')$ and $S' \cong_\theta T'$.*

Proof: Follows from Theorem 3 and Lemmas 5 and 6. □

7. Discussion of completeness

Soundness (rejecting all incorrect code transformations) is the fundamental property of a validator; but relative completeness (not rejecting valid code transformations) is important in practice, since

a validator that reports many false positives is useless. Semantic equivalence between two arbitrary pieces of code being undecidable, a general-purpose validation algorithm cannot be sound and complete at the same time. However, a specialized validator can still be sound and complete for a specific class of program transformations—in our case, software pipelining optimizations based on modulo scheduling and modulo variable expansion. In this section, we informally discuss the sources of potential incompleteness for our validator.

Mismatch between infinitary and finitary symbolic evaluation A first source of potential incompleteness is the reduction of the infinitary condition $\forall N,\ \alpha(X_N) = \alpha(Y_N)$ to the two finitary checks A and B. More precisely, the soundness proof of section 6 hinges on the following implication being true:

$$(A) \wedge (B) \Longrightarrow (17)$$

or in other words,

$$\alpha(\mathcal{B}^{\mu}) \approx_{\theta} \alpha(\mathcal{P};\mathcal{E}) \ \wedge\ \alpha(\mathcal{E};\mathcal{B}^{\delta}) \approx_{\theta} \alpha(\mathcal{S};\mathcal{E}) \\ \Longrightarrow\ \forall n, \alpha(\mathcal{B}^{\mu+n\delta}) \approx_{\theta} \alpha(\mathcal{P};\mathcal{S}^{n};\mathcal{E})$$

However, the reverse implication does not hold in general. Assume that (17) holds. Taking $n = 0$, we obtain equation (A). However, we cannot prove (B). Exploiting (17) for n and $n+1$ iterations, we obtain:

$$\alpha(\mathcal{B}^{\mu+n\delta}) \approx_{\theta} \alpha(\mathcal{P};\mathcal{S}^{n});\alpha(\mathcal{E})$$
$$\alpha(\mathcal{B}^{\mu+n\delta});\alpha(\mathcal{B}^{\delta}) \approx_{\theta} \alpha(\mathcal{P};\mathcal{S}^{n});\alpha(\mathcal{S};\mathcal{E})$$

Combining these two equivalences, it follows that

$$\alpha(\mathcal{P};\mathcal{S}^{n});\alpha(\mathcal{E};\mathcal{B}^{\delta}) \approx_{\theta} \alpha(\mathcal{P};\mathcal{S}^{n});\alpha(\mathcal{S};\mathcal{E}) \qquad (26)$$

However, we cannot just "simplify on the left" and conclude $\alpha(\mathcal{E};\mathcal{B}^{\delta}) \approx_{\theta} \alpha(\mathcal{S};\mathcal{E})$. To see why such simplifications are incorrect, consider the two blocks

$$\begin{array}{ll} x := 1; & \text{and} \quad x := 1; \\ y := 1 & \qquad\ \ y := x \end{array}$$

We have $\alpha(x := 1);\alpha(y := 1) \approx_{\theta} \alpha(x := 1);\alpha(y := x)$ with $\theta = \{x, y\}$. However, the equivalence $\alpha(y := 1) \approx_{\theta} \alpha(y := x)$ does not hold.

Coming back to equation (26), the blocks $\mathcal{E};\mathcal{B}^{\delta}$ and $\mathcal{S};\mathcal{E}$ could make use of some property of the state set up by $\mathcal{P};\mathcal{S}^{n}$ (such as the fact that $x = 1$ in the simplified example above). However, (26) holds for any number n of iteration. Therefore, this hypothetical property of the state that invalidates (B) must be a loop invariant. We are not aware of any software pipelining algorithm that takes advantage of loop invariants such as $x = 1$ in the simplified example above.

In conclusion, the reverse implication $(17) \Longrightarrow (A) \wedge (B)$, which is necessary for completeness, is not true in general, but we strongly believe it holds in practice as long as the software pipelining algorithm used is purely syntactic and does not exploit loop invariants.

Mismatch between symbolic evaluation and concrete executions A second, more serious source of incompleteness is the following: equivalence between the symbolic evaluations of two basic blocks is a sufficient, but not necessary condition for those two blocks to execute identically. Here are some counter-examples.

1. The two blocks could execute identically because of some property of the initial state, as in

   $$y := 1 \quad \text{vs.} \quad y := x$$

   assuming $x = 1$ initially.

2. Symbolic evaluation fails to account for some algebraic properties of arithmetic operators:

   $$\begin{array}{ll} i := i + 1; & \text{vs.} \quad i := i + 2 \\ i := i + 1 & \end{array}$$

3. Symbolic evaluation fails to account for memory separation properties within the same memory block:

   $$\begin{array}{ll} \texttt{store}(\texttt{int32}, p+4, y); & \text{vs.} \quad x := \texttt{load}(\texttt{int32}, p); \\ x := \texttt{load}(\texttt{int32}, p) & \qquad\ \ \texttt{store}(\texttt{int32}, p+4, y) \end{array}$$

4. Symbolic evaluation fails to account for memory separation properties within different memory blocks:

   $$\begin{array}{ll} \texttt{store}(\texttt{int32}, a, y); & \text{vs.} \quad x := \texttt{load}(\texttt{int32}, b); \\ x := \texttt{load}(\texttt{int32}, b) & \qquad\ \ \texttt{store}(\texttt{int32}, a, y) \end{array}$$

   assuming that a and b point to different arrays.

Mismatches caused by specific properties of the initial state, as in example 1 above, are not a concern for us: as previously argued, it is unlikely that the pipelining algorithm would take advantage of these properties. The other three examples are more problematic: software pipeliners do perform some algebraic simplifications (mostly, on additions occurring within address computations and loop index updates) and do take advantage of nonaliasing properties to hoist loads above stores.

Symbolic evaluation can be enhanced in two ways to address these issues. The first way, pioneered by Necula [18], consists in comparing symbolic terms and states modulo an *equational theory* capturing the algebraic identities and "good variable" properties of interest. For example, the following equational theory addresses the issues illustrated in examples 2 and 3:

$$\texttt{Op}(\texttt{addi}(n), \texttt{Op}(\texttt{addi}(m), t)) \equiv \texttt{Op}(\texttt{addi}(n+m), t)$$
$$\texttt{Load}(\kappa', t_1, \texttt{Store}(\kappa, t_2, t_v, t_m)) \equiv \texttt{Load}(\kappa', t_2, t_m) \\ \text{if } |t_2 - t_1| \geq \texttt{sizeof}(\kappa)$$

The separation condition between the symbolic addresses t_1 and t_2 can be statically approximated using a decision procedure such as Pugh's Omega test [21]. In practice, a coarse approximation, restricted to the case where t_1 and t_2 differ by a compile-time constant, should work well for software pipelining.

A different extension of symbolic evaluation is needed to account for nonaliasing properties between separate arrays or memory blocks, as in example 4. Assume that these nonaliasing properties are represented by region annotations on load and store instructions. We can, then, replace the single resource $\texttt{Mem}$ that symbolically represent the memory state by a family of resources $\texttt{Mem}^{x}$ indexed by region identifiers x. A load or store in region x is, then, symbolically evaluated in terms of the resource $\texttt{Mem}^{x}$. Continuing example 4 above and assuming that the store takes place in region a and the load in region b, both blocks symbolically evaluate to

$$\texttt{Mem}^{a} \mapsto \texttt{Store}(\texttt{int32}, a_0, y_0, \texttt{Mem}^{a}_{0})$$
$$x \mapsto \texttt{Load}(\texttt{int32}, b_0, \texttt{Mem}^{b}_{0})$$

8. Implementation and preliminary experiments

The first author implemented the validation algorithm presented here as well as a reference software pipeliner. Both components are implemented in OCaml and were connected with the CompCert C compiler for testing.

The software pipeliner accounts for approximately 2000 lines of OCaml code. It uses a backtracking iterative modulo scheduler [2, chapter 20] to produce a steady state. Modulo variable expansion is then performed following Lam's approach [10]. The modulo scheduler uses a heuristic function to decide the order in which

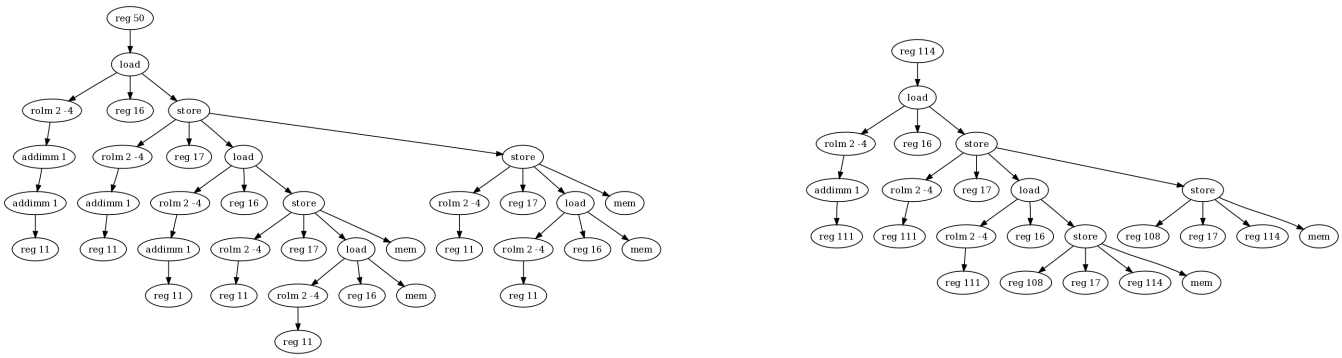


Figure 6. The symbolic values assigned to register 50 (left) and to its pipelined counterpart, register 114 (right). The pipelined value lacks one iteration due to a corner case error in the implementation of the modulo variable expansion. Symbolic evaluation was performed by omitting the prolog and epilog moves, thus showing the difference in register use.

to consider the instructions for scheduling. We tested our software pipeliner with two such heuristics. The first one randomly picks the nodes. We use it principally to stress the pipeliner and the validator. The second one picks the node using classical dependencies constraints. We also made a few schedules by hand.

We experimented our pipeliner on the CompCert benchmark suite, which contains, among others, some numerical kernels such as a fast Fourier transform. On these examples, the pipeliner performs significant code rearrangements, although the observed speedups are modest. There are several reasons for this: we do not exploit the results of an alias analysis; our heuristics are not state of the art; and we ran our programs on an out-of-order PowerPC G5 processor, while software pipelining is most effective on in-order processors.

The implementation of the validator follows very closely the algorithm presented in this paper. It accounts for approximately 300 lines of OCaml code. The validator is instrumented to produce a lot of debug information, including in the form of drawings of symbolic states. The validator found many bugs during the development of the software pipeliner, especially in the implementation of modulo variable expansion. The debugging information produced by the validator was an invaluable tool to understand and correct these mistakes. Figure 6 shows the diagrams produced by our validator for one of these bugs. We would have been unable to get the pipeliner right without the help of the validator.

On our experiments, the validator reported no false alarms. The running time of the validator is negligible compared with that of the software pipeliner itself (less than 5%). However, our software pipeliner is not state-of-the-art concerning the determination of the minimal initiation interval, and therefore is relatively costly in terms of compilation time.

Concerning formal verification, the underlying theory of symbolic evaluation (section 4) and the soundness proof with respect to unrolled loops (section 6.1) were mechanized using the Coq proof assistant. The Coq proof is not small (approximately 3500 lines) but is relatively straightforward: the mechanization raised no unexpected difficulties. The only part of the soundness proof that we have not mechanized yet is the equivalence between unrolled loops and their CFG representations (section 6.2): for such an intuitively obvious result, a mechanized proof is infuriatingly difficult. Simply proving that the situation depicted in figure 2(a) holds (we have a proper, counted loop whose body is the basic block $\mathcal{B}$) takes a great deal of bureaucratic formalization. We suspect that the CFG representation is not appropriate for this kind of proofs; see section 10 for a discussion.

9. Related work

This work is not the first attempt to validate software pipelining *a posteriori*. Leviathan and Pnueli [14] present a validator for a software pipeliner for the IA-64 architecture. The pipeliner makes use of a rotating register file and predicate registers, but from a validation point of view it is almost the same problem as the one studied in this paper. Using symbolic evaluation, the validator generates a set of verification conditions that are discharged by a theorem prover. We chose to go further with symbolic evaluation: instead of using it to generate verification conditions, we use it to directly establish semantic preservation. We believe that the resulting validator is simpler and algorithmically more efficient. However, exploiting nonaliasing information during validation could possibly be easier in Leviathan and Pnueli's approach than in our approach.

Another attempt at validating software pipelining is the one of Kundu *et al.* [9]. It proceeds by parametrized translation validation. The software pipelining algorithm they consider is based on code motion, a rarely-used approach very different from the modulo scheduling algorithms we consider. This change of algorithms makes the validation problem rather different. In the case of Kundu *et al.* , the software pipeliner proceeds by repeated rewriting of the original loop, with each rewriting step being validated. In contrast, modulo scheduling builds a pipelined loop in one shot (usually using a backtracking algorithm). It could possibly be viewed as a sequence of rewrites but this would be less efficient than our solution.

Another approach to establishing trust in software pipelining is run-time validation, as proposed by Goldberg *et al.* [5]. Their approach takes advantages of the IA-64 architecture to instrument the pipelined loop with run-time check that verify properties of the pipelined loop and recover from unexpected problems. This technique was used to verify that aliasing is not violated by instruction switching, but they did not verify full semantic preservation.

10. Conclusions

Software pipelining is one of the pinnacles of compiler optimizations; yet, its semantic correctness boils down to two simple semantic equivalence properties between basic blocks. Building on this novel insight, we presented a validation algorithm that is simple enough to be used as a debugging tool when implementing software pipelining, efficient enough to be integrated in production compilers and executed at every compilation run, and mathematically elegant enough to lend itself to formal verification.

Despite its respectable age, symbolic evaluation—the key technique behind our validator—finds here an unexpected application

to the verification of a loop transformation. Several known extensions of symbolic evaluation can be integrated to enhance the precision of the validator. One, discussed in section 7, is the addition of an equational theory and of region annotations to take nonaliasing information into account. Another extension would be to perform symbolic evaluation over extended basic blocks (acyclic regions of the CFG) instead of basic blocks, like Rival [24] and Tristan and Leroy [26] do. This extension could make it possible to validate pipelined loops that contain predicated instructions or even conditional branches.

From a formal verification standpoint, symbolic evaluation lends itself well to mechanization and gives a pleasant denotational flavor to proofs of semantic equivalence. This stands in sharp contrast with the last step of our proof (section 6.2), where the low-level operational nature of CFG semantics comes back to haunt us. We are led to suspect that control-flow graphs are a poor representation to reason over loop transformations, and to wish for an alternate representation that would abstract the syntactic details of loops just as symbolic evaluation abstracts the syntactic details of (extended) basic blocks. Two promising candidates are the PEG representation of Tate *et al.* [25] and the representation as sets of mutually recursive HOL functions of Myreen and Gordon [17]. It would be interesting to reformulate software pipelining validation in terms of these representations.

Acknowledgments

This work was supported by Agence Nationale de la Recherche, project U3CAT, program ANR-09-ARPEGE. We thank François Pottier, Alexandre Pilkiewicz, and the anonymous reviewers for their careful reading and suggestions for improvements.

References

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, second edition, 2006.

[2] A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[3] C. W. Barret, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification, 17th Int. Conf., CAV 2005*, volume 3576 of *LNCS*, pages 291–295. Springer, 2005.

[4] J. M. Codina, J. Llosa, and A. González. A comparative study of modulo scheduling techniques. In *Proc. of the 16th international conference on Supercomputing*, pages 97–106. ACM, 2002.

[5] B. Goldberg, E. Chapman, C. Huneycutt, and K. Palem. Software bubbles: Using predication to compensate for aliasing in software pipelines. In *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT 2002)*, pages 211–221. IEEE Computer Society Press, 2002.

[6] Y. Huang, B. R. Childers, and M. L. Soffa. Catching and identifying bugs in register allocation. In *Static Analysis, 13th Int. Symp., SAS 2006*, volume 4134 of *LNCS*, pages 281–300. Springer, 2006.

[7] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267. ACM, 1993.

[8] A. Kanade, A. Sanyal, and U. Khedker. A PVS based framework for validating compiler optimizations. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 108–117. IEEE Computer Society, 2006.

[9] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 327–337. ACM Press, 2009.

[10] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328. ACM, 1988.

[11] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[12] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 2009. To appear.

[13] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[14] R. Leviathan and A. Pnueli. Validating software pipelining optimizations. In *Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2002)*, pages 280–287. ACM Press, 2006.

[15] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, 1996.

[16] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[17] M. O. Myreen and M. J. C. Gordon. Transforming programs into recursive functions. In *Brazilian Symposium on Formal Methods (SBMF 2008)*, volume 240 of *ENTCS*, pages 185–200. Elsevier, 2009.

[18] G. C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.

[19] 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*, 2(2):192–201, 1998.

[20] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[21] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[22] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Processing*, 24(1):1–102, 1996.

[23] B. R. Rau, M. S. Schlansker, and P. P. Timmalai. Code generation schema for modulo scheduled loops. Technical Report HPL-92-47, Hewlett-Packard, 1992.

[24] X. Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.

[25] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *36th symposium Principles of Programming Languages*, pages 264–276. ACM Press, 2009.

[26] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17–27. ACM Press, 2008.

[27] J.-B. Tristan and X. Leroy. Verified validation of Lazy Code Motion. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 316–326. ACM Press, 2009.

[28] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

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