Semantic Foundations for Cost Analysis of Pipeline-Optimized Programs

Gilles Barthe[1], Adrien Koutsos[2], Solène Mirliaz[3], David Pichardie[4], and Peter Schwabe[1]

[1] MPI-SP & IMDEA Software Institute, Bochum, Germany
[2] Inria Paris, France
[3] Univ Rennes, CNRS, IRISA, France
[4] Meta, France

Abstract. In this paper, we develop semantic foundations for precise cost analyses of programs running on architectures with multi-scalar pipelines and in-order execution with branch prediction. This model is then used to prove the correction of an automatic cost analysis we designed. The analysis is implemented and evaluated in an extant framework for high-assurance cryptography. In this field, developers aggressively hand-optimize their code to take maximal advantage of micro-architectural features while looking for provable semantic guarantees.

1 Introduction

Provable cost analysis, such as [29,22], provides a rich palette of methods and tools for estimating (generally in the form of upper bounds) execution time with respect to a mathematical operational and cost model. However, operational and cost models commonly used in provable cost analysis elude micro-architectural features, such as caches, predictors, and pipelines, which are performance-critical and carefully exploited in high-performance implementations. As a consequence, the upper bounds computed by existing cost analyses are overly coarse. In particular, they cannot be used to guide carefully crafted manual optimizations, for instance the instruction scheduling of the program, since a typical provable cost analysis will be oblivious to instruction scheduling.

Specific areas of computer science require high-performance and maximal reliability. It is for example the case of cryptographic engineers who develop high-speed implementations of common cryptographic algorithms. Increasingly, cryptographic engineering is adopting high-assurance techniques [4] to deliver provable guarantees that implementations are correct with respect to their high-level specification (expressed mathematically or as pseudo-code), cryptographically secure, and protected against side-channels. Unfortunately, high-assurance cryptography still relies on simulation or benchmarking for measuring the efficiency of implementations, largely ignoring the line of work in provable cost analysis.

Listing 1.1 provide a classic example of an array sum program that can be aggressively optimized in order to take advantage of modern micro-architectural

```
r = 0;          //1
t = [A +  0]; //1
r += t;         //3
t = [A +  4]; //3
r += t;         //5
t = [A +  8]; //5
r += t;         //7
t = [A + 12]; //7
r += t;         //9
t = [A + 16]; //9
r += t;         //11
t = [A + 20]; //11
r += t;         //13
t = [A + 24]; //13
r += t;         //15
t = [A + 28]; //15
r += t;         //17
```

Listing 1.1: Straightforward

```
r0 = 0;         //1
r1 = 0;         //1
t0 = [A +  0]; //1
t1 = [A +  4]; //2
t2 = [A +  8]; //2
r0 += t0;       //3
t0 = [A + 12]; //3
r1 += t1;       //4
t1 = [A + 16]; //4
r0 += t2;       //4
t2 = [A + 20]; //5
r1 += t0;       //5
t0 = [A + 24]; //5
r0 += t1;       //6
t1 = [A + 28]; //6
r1 += t2;       //7
r0 += t0;       //7
r1 += t1;       //8
r = r0+r1;      //9
```

Listing 1.2: Optimized

Fig. 1: Two different approaches to scheduling instructions for code that accumulates 8 consecutive 32-bit integers from memory. Comments indicate execution cycles on the microarchitecture described in Fig. 2.

mechanisms. The program computes (in variable r) the sum of the elements of an array A. An optimized version of this program is given in Listing 1.2, which exploits the architecture capability to perform loads in parallel, avoiding the two cycles penalty for each element occurring in Listing 1.1. It thus uses more registers to store the pending results. A standard cost analysis would conclude, wrongly, that the optimized program has a worst execution time than the original: indeed, both programs executed the same amount of loads, but the optimized program performs an additional assignment and addition. Summing the delay of each instruction, as a naive cost analysis would do, concludes that the optimized version is worse than the original. To understand the benefit of this optimization, the programmer has to reason on the model of instruction parallelism.

This paper develops semantic foundations for cost analysis of pipelined-optimized programs. We focus on the instruction pipeline mechanism and do not model caches in this work. Our work is intended for the programmer who wants to formally check the cost impact of manual optimizations. Such programmers are usually happy to assume that all program code and all data is in L1 cache, in order to focus on careful instruction selection, scheduling, and register allocation. Cryptographic primitives fall into this case. We focus on in-order processors, as out-of-order processors will change the scheduling imagined by the

programmer. Although out-of-order processors are more common due to their efficiency, manual optimizations are still particularly relevant for in-order embedded systems. Indeed, embedded systems cannot handle the complexity and energy cost of out-of-order processors.

Our work makes the following contributions.

- We provide a detailed semantic model, presented in Section 3, which is a small-step semantics precisely modeling the execution cost (in processor cycles) of instruction parallelism and branch prediction inside an in-order processor.
- We then design in Section 4 a provably correct static analysis that computes safe relational bounds on this cost. The analysis is a mix of a standard relational numerical analysis, a standard may/must static analysis and a new block symbolic execution that extracts a tight range for the execution time of an instruction block. The static analysis is proven sound with respect to the small-step semantics (Theorem 3). The full proof of correctness is given in the companion report [6].
- We have implemented our approach into Jasmin [2,3], an existing framework for high-performance and high-assurance cryptography. We use our analysis to obtain relational cost bounds for scalar and vectorized implementations of popular cryptographic algorithms. These experiments show that our estimates are precise (in particular the difference between the upper and lower bounds is tight), and significantly improve on the bounds delivered by traditional cost analyses which ignore instruction parallelism.

2 Processor Behavior on an Example

We consider a low-level language (inspired from Jasmin [2,3] internal representation), with memory load/store, and scalar operations. Programs in our language are executed on a *multi-scalar pipelined processor*. A *pipelined processor* decomposes the execution of an atomic instruction into several stages such that the next instruction can enter the first stage as soon as the previous instruction leaves it. A sequence of stages constitutes a pipeline, and the latency of a pipeline is the number of stages it comprises. A multi-scalar pipelined processor has several pipelines in parallel, allowing it to execute simultaneously several instructions, by loading them into different pipelines. All pipelines are not identical: each pipeline can have a different latency, and supports a different set of instructions. The latency of a pipeline depends on the instructions supported, where basic instructions, such as additions, will be executed quickly, while more complex

	A	L	S	M	J
Add/Sub (1)	✓	✓			
Comp (1)	✓	✓	✓		
Load (2)		✓	✓		
Store (2)			✓		
Mult (5)				✓	
Jump (4)					✓

Fig. 2: Instructions handled by each pipeline of our processor, with their latencies in parenthesis

operations (e.g. multiplications and floating-point operations) will take a longer time.

Fig. 2 describes an example of a processor with five pipelines (A, L, S, M and J) and the instructions each pipeline can handle: for example, multiplication has a latency of 5, and is only supported by the pipeline M. This is a simple processor, real processors have more pipelines and can handle a larger instruction set. Note that the method presented in this paper is not specific to this processor: the number of pipelines, the instructions supported and their latencies are parameters of the cost semantics and of the analysis.

Instruction Fetching We now give a high-level overview of how a processor fetches an instruction, which is done in three steps. First, the processor checks that the instruction has no data-dependency conflict with other instructions already in the pipelines. Then, the processor resolves the instruction by evaluating the registers read by the instruction into values – which are either integers or memory addresses. Finally, the resolved instruction, called a *transient* instruction, is placed in a pipeline supporting it.

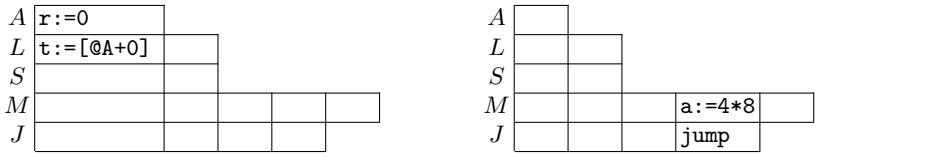
Data-dependencies Before starting executing an instruction – i.e. loading it in the first stage of a pipeline – the processor must check that this instruction has no conflict with other instructions being currently executed. For example, consider the execution of lines 1 through 3 of Listing 1.1 on the processor of Fig. 2. The resulting state of the processor can be found in Fig. 3a. The first instruction can be placed in stage A_1 (the first stage of the A pipeline), while simultaneously loading the second instruction into stage L_1. However, the instruction of the third line cannot be loaded during the same cycle, because it depends on the values of registers `r` and `t`, which will be written by the previous instructions: the processor must wait for their executions to finish before fetching l.3.

Essentially, an instruction can be executed if: i) there is a pipeline available (i.e. whose first stage is empty) supporting it; and ii), none of its variables (a.k.a. registers or memory locations such as `@A`) have *data-dependencies* with instructions currently in the pipelines. More precisely, an instruction `atom` cannot be executed if:

- any variable it reads is written by another instruction currently in a pipeline (*read-after-write* dependency);
- any variable it writes is read or written by another instruction in the pipeline (*write-after-read* and *write-after-write*).

We refer to these dependencies using the acronyms RaW, WaR and WaW. Coming back to our example, the instruction l.3 needs to wait for two cycles – the latency of the load – to be fetched after l.2 because of a RaW dependency on `t`.

Instruction Resolution Before being placed in the first stage of a pipeline supporting it, the instruction is *resolved*, by replacing the registers it reads by their current value. We illustrate this mechanism on the array sum (Listing 1.1). Let us suppose that the first cell of A contains value 32, stored in `t` after the execution



(a) State of the pipelines after line 5 and 6 of the first iteration of Listing 1.1

(b) State of the pipelines after fetching a jump

Fig. 3: Example of pipeline states for the processor of Fig. 2. Each cell represents a pipeline stage, e.g. stage J_4 in the second state contains a jump.

of l.2. The instruction l.3 `r := r + t` is resolved into the transient instruction `r := 0 + 32`. Note that a transient instruction no longer reads any register, which allows to avoid some data-dependency conflicts. After the instruction l.2 has been fetched, we can expect the pipelines to be in the state of Fig. 3a, where `@A` designates the address stored in A.

Branch prediction When the processor executes a sequence, it simply increments its program counter to find the next instruction to execute. But in the case of a conditional jump, the next instruction to execute is harder to infer. In that case, a jump must be resolved: if the jump is taken, then its destination is computed and used to update the program counter. Otherwise, the processor continues its execution with an incremented program pointer. The jump must go through all the stages of its pipeline to affect the program counter. Not fetching any instruction during its processing would severely impact the performances of the processor. It is more interesting to start fetching and executing one of the two branches as soon as a jump is encountered, without waiting for the jump to be fully processed. The branch predictor (BP) is in charge of deciding which branch will be speculatively executed. It typically uses a history, usually in the form of a buffer, to remember the previous branches taken and bases its decisions upon it. When the jump has been fully processed, the prediction is checked. In case of a correct prediction, the execution of the speculated branch continues. Otherwise, all the modifications made by the speculated branch must be roll backed, and the correct branch starts its execution. The roll-back requires to buffer the speculated instructions when they are retired from their pipeline and to identify which instructions in the pipelines are speculation.

The content of the pipelines, i.e. the instructions already loaded, is not sufficient to roll back the pipelines. For example, consider the following two code snippets. The instruction $\texttt{jmp}(c) : T$ is a conditional jump: the program continues with the instruction at address T – further in the code – if c holds, or goes to the next instruction otherwise. So the *then* branch of this conditional is not displayed here, only its *else* branch. In the first code snippet, the *else* branch contains only l.3, while it contains l.2-3 in the second.

```
1 a := 4 * 8;
2 jump (c) : T;
3 b := 2 + 6;
```

and

```
1 jump (c) : T;
2 a := 4 * 8;
3 b := 2 + 6;
```

These two programs are executed from empty pipelines and we assume here that the *else* branch is speculatively executed. Let us take a snapshot of the processor state after the three instructions have been fetched and after the processor has executed three cycles to make the instructions progress in their pipelines. For both executions, the pipelines should be in the state of Fig. 3b. Notice that the speculated addition `b := 2 + 6` has been fully executed and has left the pipeline. Also, in both cases, the multiplication is at the same depth (4) as the jump, and there is no way of telling if it was speculatively executed, or if it was fetched before the jump. Hence it is not possible to determine if the multiplication must be removed simply by inspecting the pipelines.

Therefore, to be able to perform roll backs, the processor: (i) buffers the effects of the retired instructions (here the addition); and (ii), timestamps the instructions to track their dependencies. Any instruction that has been fully executed is placed into a buffer, called the *speculation buffer*, before acting on the memory. Once it is guaranteed that no previous jump can roll it back, it is *committed*, effectively modifying the memory. When a roll back is performed, any instruction in the buffer or the pipelines with an higher timestamp than the jump is removed. These mechanisms are inspired from [10].

3 Concrete Small-step Pipeline Semantics

In this section we define the concrete small-step semantics of a multi-pipelined processor where the cost in cycles is tracked. This semantics precisely models a pipelined processor with branch prediction. It includes a speculation buffer in order to model the roll back mechanism used after branch misprediction. In the next section, we will present an approximation of this semantics w.r.t. the cost, which we use to build a sound static analysis. Fig. 5 summarizes the notations used by our semantics rules in Fig. 7, 8 and 9.

Language The syntax of our language is given in Fig. 4. Atomic instructions $\mathtt{atom} \in \mathsf{Atoms}$ can be basic arithmetic operations, memory loads/stores and jump instructions. The instructions operate on registers in Reg, which can contain integer values in $\mathbb{Z}$ or memory locations in $\mathsf{MemLocs}$. Finally, programs are built using sequential composition of atomic instructions, conditionals and while loops. The jump instruction is not meant to be directly written by the programmer. Its role will be explained in the semantic rules for conditionals. Conditionals and loops are annotated with distinct labels ℓ in the set of labels $\mathcal{L}$. The branch predictor uses them to distinguish the different conditional jumps and to build its history of past jumps.

The syntax is inspired from the Jasmin language [2,3], which features precisely such a combination of low-level atomic instructions that translate directly to assembly and high-level structures consisting of while loops and conditionals.

Operands:

o	$::= r \in \mathsf{Reg}$	Register	
	$\mid n \in \mathbb{Z}$	Integer	

Labels:

ℓ	$\in \mathcal{L}$	

Atomic instructions Atoms:

$\texttt{atom}$	$::= r := o_1 + o_2$	Addition	
	$\mid r := o_1 - o_2$	Subtraction	
	$\mid r := o_1 \leq o_2$	Comparison	
	$\mid r := o_1 \times o_2$	Multiplication	
	$\mid r_1 := [r_2 + o]$	Load	
	$\mid [r + o_1] := o_2$	Store	
	$\mid \texttt{jmp}(o)$	Conditional jump	

Statements:

s	$::= \texttt{atom}$		Atomic
	$\mid s_1; s_2$		Sequence
	$\mid \ell : \texttt{if } o \texttt{ then } s_1$		
		$\texttt{else } s_2$	Conditional
	$\mid \ell : \texttt{while } o$		
		$\texttt{do } s \texttt{ done}$	Loop
	$\mid \texttt{skip}$		Skip

Fig. 4: Syntax of the language

Memory State Values are stored at locations, $\mathsf{Location} = \mathsf{Reg} \cup \mathsf{MemLocs}$, comprising registers and memory locations. A memory state $\sigma : \mathsf{Location} \mapsto \mathsf{Val}$ is a map from locations to values, which are either integers or memory locations (see Fig. 5). For any atomic instruction $\texttt{atom}$ and memory state σ, we let $\mathbb{S}[\![\texttt{atom}]\!]\sigma$ be the memory state obtained when evaluating $\texttt{atom}$ in σ. This atomic instruction semantics is defined as usual — we omit the details.

Pipeline State Our semantics is parametric in the processor's architecture, i.e. the number of pipelines, the instructions they support, and the instructions' latencies. For simplicity, the jump instruction is handled by a single pipeline J. This is the usual settings for branch predictors as it simplifies the design of the processor. Formally, we assume a fixed set of pipelines Pips. For every pipeline $X \in \mathsf{Pips}$, we note X_i the i-th stage of X. For any atomic instruction $\texttt{atom}$, its latency characterizes the number of stages required to execute the instruction before it can leave the pipeline. We note $|\texttt{atom}|$ its latency, and we write $X \in \texttt{atom}$ if the pipeline X handles the instruction $\texttt{atom}$. We also confuse $\texttt{atom}$ with the set of *all* pipelines that handle $\texttt{atom}$. Then, the latency of a pipeline $|X|$ is the maximal latency of the instructions it supports. The pipelines are ordered so that given an instruction handled by several pipelines, these pipelines will be checked in a fixed order. For instance on our processor, for a comparison, the pipelines will be checked in the order A, then L, then S. As a shorthand, we write $X = \min\{Y \in \texttt{atom}\}$ to get the first pipeline handling $\texttt{atom}$.

Each stage of a pipeline is either empty (denoted ϵ), or contains a transient instruction – obtained by resolving an atomic instruction – ready to be processed. The set of transient instructions is denoted $\mathsf{Atoms}_\mathsf{t}$. As explained in Section 2, we need to annotate the instructions in the pipelines to know if they are speculation and depend on a jump retiring. Each transient instruction in a pipeline stage is associated to a timestamp, which orders it w.r.t. the other instructions in the pipelines. A smaller timestamp denotes an older instruction. The timestamp is

Latency

$\lvert\texttt{atom}\rvert$	$\in \mathbb{N}$	

Values (Val) :

v	$::= l \in \mathsf{MemLocs}$	Memory location
	$\mid n \in \mathbb{Z}$	Number

Locations ($\mathsf{Location}$):

x	$::= l \in \mathsf{MemLocs}$	Memory location
	$\mid r \in \mathsf{Reg}$	Register

Memory state (S):

σ	$\in \mathsf{Location} \to \mathsf{Val}$	

Pipelines:

X	$\in \mathsf{Pips}$	Pipeline
$X_1, X_2, \ldots$	$\in \mathsf{Stages}$	Stage
ϵ		Empty stage content

Transient instructions ($\mathsf{Atoms_t}$):

$\texttt{atom}_\mathsf{t}$	$::= r := v_1 \bowtie v_2$	Scalar operations ($\bowtie \in \{+, -, \times, \leq\}$)
	$\mid r := [l + n]$	Load
	$\mid [l + n] := v$	Store
	$\mid \texttt{jmp}(v)$	Jump

Pipeline state:

Cells	$= ((\mathbb{N} \times \mathsf{Atoms_t}) \cup \epsilon)$	Cells
π	$\in \mathsf{Stages} \to \mathsf{Cells}$	Pipeline state
$\pi[j : j \leq i]$		Roll back of instructions older than i

Branch prediction (BP):

h		Branch prediction history
BP-predict (h, ℓ)		BP prediction on jump ℓ
BP-update $(h, \ell, taken)$		Update the BP history with jump results

Speculation buffer:

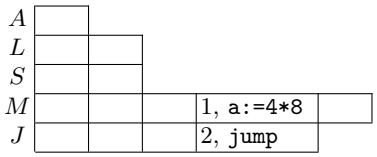
β	$\in \mathcal{P}(\mathbb{N} \times \mathsf{Atoms_t})$	Speculation buffer
$\min(\beta, \pi)$	$\in \mathbb{N}$	Minimal index in β and π ($= 0$ if empty)
$\max(\beta, \pi)$	$\in \mathbb{N}$	Maximal index in β and π ($= 0$ if empty)
$\beta(\sigma)$	$= (\bigcirc_{(j,\texttt{atom}_\mathsf{t}) \in \beta} \mathbb{S}[\![\texttt{atom}_\mathsf{t}]\!])(\sigma)$	Application of all instructions of β
$\beta[j : j \leq i]$	$\in \mathcal{P}(\mathbb{N} \times \mathsf{Atoms_t})$	All instructions more recent than i

Processor state:

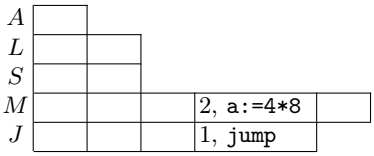
ω	$= \langle \sigma, \pi, h, \beta \rangle$	Processor state

Fig. 5: Concrete pipelined processor

incremented each time we fetch a new instruction. Therefore, a pipeline state π is a function from pipeline stages Stages to pairs of an integer and a transient instruction ($(i, \texttt{atom}_\mathsf{t}) \in (\mathbb{N} \times \mathsf{Atoms_t})$), or to the empty slot ϵ. To be able to roll back a jump with index i, we use the pipeline state $\pi[j : j \leq i]$, which is the state π where only instructions older than i in π have been kept. Newer instructions of π (i.e. such that $\pi(X_k) = (j, \texttt{atom}_\mathsf{t})$ with $j > i$) are replaced with ϵ. We illustrate this in Fig. 6, using the branch prediction example of Section 2. Recall that the



(a) The jump has been fetched after the assignment



(b) The jump has been fetched before the assignment, and thus depends on its prediction

Fig. 6: The timestamps associated to the instructions records prediction dependencies, and allow to perform roll backs if necessary.

two programs had the same pipelines state (described in Fig. 3b). But when adding the timestamps, we obtain two distinct states. In the first case (Fig. 6a), the multiplication has been fetched before the jump, and thus its timestamps (1) is smaller than the one of the jump (2). Hence, in case of rollback due to a misprediction of the jump, the multiplication will not be evinced. In the second case (Fig. 6b), the multiplication is speculatively executed, and fetched after the jump: its timestamps (2) is greater than the one of the jump (1), and will thus be evinced if the jump destination was mispredicted.

Speculation buffer After it has been executed, an instruction is stored in the speculation buffer β. The instruction will be committed, i.e. its effect will be applied on the memory σ, only when the processor is guaranteed that it was not an incorrect speculation. Similarly to the pipeline state π, the speculation buffer β keeps track of the index of the instructions to check the sequential dependencies. Hence β is a set of pairs $(i, \mathtt{atom_t}) \in (\mathbb{N} \times \mathsf{Atoms_t})$. We let $\min(\beta, \pi)$ be the minimal index associated to an instruction in β and π (we define similarly $\max(\beta, \pi)$). Similarly to π, $\beta[j : j \leq i]$ is the buffer β where only the instructions older than i in β have been kept. The effect of the instructions in the speculation buffer should be taken into account as if it was already applied on the memory state σ. The notation $\beta(\sigma)$ corresponds to the application on σ of these instructions, from the oldest to the most recent.

Branch prediction history The branch predictor is guided by a history of previous jumps. Usually, it is a buffer associating a boolean *taken* or *not taken* to each jump label ℓ, but this can change depending on the processor. Therefore, we chose to keep its precise implementation abstract in our model. We note h this history and assume two operators: BP-predict(h, ℓ) holds if the BP predicts that the jump at ℓ will be taken; and $h' =$ BP-update$(h, \ell, taken)$ updates the history depending on whether or not the jump was actually taken. We suppose that these operations are deterministic and that the history is not modified by external sources. However, we make no assumption on the quality of the prediction: it can mispredict every time for instance.

$$\textsc{Lock RaW}\ \frac{x \in \mathrm{read}(\mathtt{atom}, \sigma) \qquad x \in \mathrm{write}(\mathtt{atom}')}{\mathrm{locks}(\mathtt{atom}, \mathtt{atom}', \sigma)} \qquad \textsc{Lock WaW}\ \frac{x \in \mathrm{write}(\mathtt{atom}, \sigma) \qquad x \in \mathrm{write}(\mathtt{atom}')}{\mathrm{locks}(\mathtt{atom}, \mathtt{atom}', \sigma)}$$

$$\textsc{Lock WaR}\ \frac{x \in \mathrm{write}(\mathtt{atom}, \sigma) \qquad x \in \mathrm{read}(\mathtt{atom}')}{\mathrm{locks}(\mathtt{atom}, \mathtt{atom}', \sigma)} \qquad \textsc{Jump lock}\ \frac{}{\mathrm{locks}(\mathtt{jmp}(\_), \mathtt{jmp}(\_), \_)}$$

Fig. 7: Rules of data dependency locks

Directives The processor behaves greedily, and tries to fetch as many instructions as possible per cycle. If no pipeline is available for the next instruction `atom`, or if `atom` has a data-dependency conflict with the instructions already in the pipelines, then the processor cannot fetch the instruction `atom` and must execute a cycle. Executing a cycle makes all instructions progress one stage further in their pipeline. When an instruction `atom` has been through $|\mathtt{atom}|$ stages, then it is retired and it is placed in the speculation buffer β. At each cycle, β tries to commit its oldest instructions.

These three actions, fetching an instruction, executing a cycle and committing from the speculation buffer, are called *directives*. The fetch `atom` directive loads the instruction `atom` in the first stage of an available pipeline. The commit directive removes the oldest instruction of the speculation buffer if it does not depend on a jump in π. Finally the cycle directive executes a processor cycle, which makes instructions progress in their pipelines, then calls directive commit. All those directives are defined by the rules in Fig. 8, and described below. Notice that the fetch directive does no need the speculation buffer β because it will always be applied on a memory state $\beta(\sigma)$.

Data-Dependencies An instruction is fetched only if the variables it reads or writes are available. This is checked by the $\mathrm{locks}(\mathtt{atom}, \mathtt{atom}', \sigma)$ statement (defined in Fig. 7), which holds whenever the instruction `atom` *has* a data dependency with the transient instruction $\mathtt{atom}'$ in the memory state σ. There are three rules — for the WaW, WaR and RaW dependencies — which are defined using the variables used by `atom`. These rules rely on the auxiliary functions $\mathrm{read}(\mathtt{atom}, \sigma)$ and $\mathrm{write}(\mathtt{atom}, \sigma)$ which return, respectively, the variables read and written by `atom` in σ — the state σ is used to check if memory accesses are in conflict. For instance, the atomic instruction $a := [b + n]$ reads the value in the memory location pointed by $b + n$, that is the memory location $\sigma(b) + n$. The functions read and write are overloaded to also compute the variables read and written by transient instructions such as $\mathtt{atom}'$: $\mathrm{read}(\mathtt{atom}')$. In that case, we do not need the memory state because transient instructions have already been resolved.

Jumps are interdependent, and we cannot fetch a jump if one is already being processed. This is captured by the Jump lock rule.

$$\mathrm{next}(\pi, X_i) = \begin{cases} \epsilon & \text{if } i = 1 \text{ or } |\pi(X_{i-1})| = i - 1 \\ \pi(X_{i-1}) & \text{otherwise} \end{cases}$$

$$\mathrm{retired}(\pi) = \{(k, \mathtt{atom_t}) \mid \exists X_i \in \mathsf{Stages}, \pi(X_i) = (k, \mathtt{atom_t}) \wedge |\mathtt{atom_t}| = i\}$$

$$\textsc{Fetch}\ \frac{X = \min\{Y \in \mathtt{atom} \mid \pi(Y_1) = \epsilon\} \quad \pi' = \pi[X_1 \mapsto (i, \mathrm{resolve}(\mathtt{atom}, \sigma))]}{(\sigma, \pi) \xhookrightarrow[\mathsf{fetch}\ (i, \mathtt{atom})]{} \pi'}$$

$$\textsc{Ready}\ \frac{\forall Y_i, \pi(Y_i) \neq \epsilon \Rightarrow \neg\, \mathrm{locks}(\mathtt{atom}, \pi(Y_i), \sigma) \quad X \in \mathtt{atom} \quad \pi(X_1) = \epsilon}{\mathrm{ready}(\mathtt{atom}, \sigma, \pi)}$$

$$\textsc{Commit}\ \frac{i = \min(\beta, \pi) \quad (i, \mathtt{atom_t}) \in \beta \quad \beta' = \beta \setminus (i, \mathtt{atom_t})}{(\sigma, \pi, \beta) \xhookrightarrow[\mathsf{commit}]{} (\mathbb{S}[\![\mathtt{atom_t}]\!]\sigma, \beta')}$$

$$\textsc{One-Cycle}\ \frac{\pi' = \pi[\forall X_i, X_i \mapsto \mathrm{next}(\pi, X_i)] \quad (\sigma, \pi', \beta \cup \mathrm{retired}(\pi)) \xhookrightarrow[\mathsf{commit}]{} {}^*(\sigma', \beta') \quad i = \min(\beta', \pi') \quad \min(\beta') \neq i}{(\sigma, \pi, \beta) \hookrightarrow (\sigma', \pi', \beta')}$$

Fig. 8: Directives in a speculative context

Fetch The Fetch rule in Fig. 8 defines the judgment $(\sigma, \pi) \xhookrightarrow[\mathsf{fetch}\ (i, \mathtt{atom})]{} \pi'$, which places an instruction in the pipelines. First, it resolves the instruction using $\mathrm{resolve}(\mathtt{atom}, \sigma)$, and then places it into the first stage of a pipeline supporting it. This fetch directive will only be applied on a state (σ, π) which does not violate the data-dependencies. This condition will be checked using the statement $\mathrm{ready}(\mathtt{atom}, \sigma, \pi)$ defined by the Ready rule, which verifies that: 1) the state (σ, π) is ready to fetch the instruction $\mathtt{atom}$, by checking that $\neg\, \mathrm{locks}(\mathtt{atom}, \mathtt{atom}', \sigma)$ for any $\mathtt{atom}'$ in the pipelines (i.e. there are no data-dependencies); and 2), that there is an available pipeline X supporting the instruction. Notice that the fetch directive does not check ready itself.

Commit The buffer β prevents mis-speculated instructions from being applied on the memory state σ. Instructions in β are committed only if they are the oldest, *i.e.* have the smallest timestamp, ensuring that they do not depend on a jump, which would then have a smaller timestamp while still being in π. This is captured by the judgment $(\sigma, \pi, \beta) \xhookrightarrow[\mathsf{commit}]{} (\sigma', \beta')$, which is defined by the Commit rule. This rule allows to commit an instruction $(i, \mathtt{atom_t})$ in the speculation buffer β if it is the oldest instruction in both the buffer and the pipeline state. Since timestamps record how old instructions are – where smaller indices denote older instructions – and since all instructions have distinct timestamps, we check that $(i, \mathtt{atom_t})$ is the oldest instruction by verifying that i is the smallest timestamp in both β and π.

Executing cycles $(\sigma, \pi, \beta) \hookrightarrow (\sigma', \pi', \beta')$ represents the execution of one cycle and is defined by the One-Cycle rule. It makes all the instructions progress

$(s, \omega) \to^t (s', \omega')$
execute t cycles and fetch as much instructions of $s \neq \texttt{skip}$ as possible before each cycle

$$\textsc{Atomic}\quad \frac{i = \max(\beta, \pi) + 1 \qquad \mathrm{ready}(\texttt{atom}, \beta(\sigma), \pi) \qquad (\beta(\sigma), \pi) \xhookrightarrow[\text{fetch } (i, \texttt{atom})]{} \pi'}{(\texttt{atom}; s, \langle \sigma, \pi, h, \beta \rangle) \to^0 (s, \langle \sigma, \pi', h, \beta \rangle)}$$

$$\textsc{Cycle}\quad \frac{\neg\, \mathrm{ready}(\texttt{atom}, \beta(\sigma), \pi) \qquad (\sigma, \pi, \beta) \hookrightarrow (\sigma', \pi', \beta')}{(\texttt{atom}; s, \langle \sigma, \pi, h, \beta \rangle) \to^1 (\texttt{atom}; s, \langle \sigma', \pi', h, \beta' \rangle)}$$

$$\textsc{Spec-Cond-True-Correct}\quad \frac{(\texttt{jmp}(b); \texttt{skip}, \omega) \to^t (\texttt{skip}, \langle \sigma_2, \pi_2, h, \beta_2 \rangle) \qquad \pi_2(J_1) = (\_, \texttt{jmp} : v) \qquad v \neq 0 \qquad \neg\, \text{BP-predict}(\ell, h) \qquad h' = \text{BP-update}(\ell, h, \mathit{false}) \qquad (s_1; s_3, \langle \sigma_2, \pi_2, h, \beta_2 \rangle) \xrightarrow{=}{}^{|\texttt{jmp}|} (s', \langle \sigma_3, \pi_3, h, \beta_3 \rangle)}{(\ell : \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2; s_3, \omega) \to^{t+|\texttt{jmp}|} (s', \langle \sigma_3, \pi_3, h', \beta_3 \rangle)}$$

$$\textsc{Spec-Cond-True-Incorrect}\quad \frac{(\texttt{jmp}(b); \texttt{skip}, \omega) \to^t (\texttt{skip}, \langle \sigma_2, \pi_2, h, \beta_2 \rangle) \qquad \pi_2(J_1) = (k, \texttt{jmp} : v) \qquad v \neq 0 \qquad \text{BP-predict}(\ell, h) \qquad h' = \text{BP-update}(\ell, h, \mathit{false}) \qquad (s_2; s_3, \langle \sigma_2, \pi_2, h, \beta_2 \rangle) \xrightarrow{=}{}^{|\texttt{jmp}|} (\_, \langle \sigma_3, \pi_3, h, \beta_3 \rangle)}{(\ell : \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2; s_3, \omega) \to^{t+|\texttt{jmp}|} (s_1; s_3, \langle \sigma_3, \pi_3[j : j \leq k], h', \beta_3[j : j \leq k] \rangle)}$$

Fig. 9: Selected small-step semantics rules with explicit speculation

one stage further in their pipeline, and relies on $\mathrm{next}(\pi, X_i)$ to get the new content of the stage X_i, according to the previous stage X_{i-1}. The operator next makes all instructions advance by one stage if they have not yet reached the end of their executions. Then, all the instructions that are retired, obtained by the operator retired, are added to β to be validated. Finally, we commit as many instructions from β as possible — we check that we no longer commit any instructions by verifying that the oldest instruction, with timestamp i, is not in the new speculation buffer β'.

Small-step Given a statement s and an initial processor state ω, the judgment $(s, \omega) \to^t (s', \omega')$ states that after t cycles of fetching and executing instructions from s, the processor ends in state ω', and it still has to fetch and execute s'. The statements s is always a sequence of the form $s_1; s_2$, and our rules are defined inductively on the syntax of s_1 — s_2 is the continuation, which is essential for the branch predictor. We describe the most important rules below, which are given in Fig. 9 — the full semantics is in Appendix B.

Atomic The rules for $s_1 = \texttt{atom}$ are Atomic and Cycle. In the Atomic rule, we test whether the current state of the processor is ready to fetch `atom` using

$(s,\omega) \xrightarrow{=}{}^{t} (s',\omega')$
execute t cycles and fetch as much instructions of s as possible before each cycle

$$\textsc{Enforce-Cycle}\ \frac{(s,\omega) \to^{t} (s',\omega')}{(s,\omega) \xrightarrow{=}{}^{t} (s',\omega')} \qquad \textsc{Enforce-Cycle-Exact}\ \frac{(s,\omega) \to^{k} (\texttt{skip},\omega'') \quad \omega'' \hookrightarrow^{t-k} \omega'}{(s,\omega) \xrightarrow{=}{}^{t} (\texttt{skip},\omega')}$$

Fig. 10: Small-step semantics to enforce arbitrary cycle execution

ready$(\texttt{atom}, \beta(\sigma), \pi)$. We use the state $\beta(\sigma)$, since an instruction to be fetched must consider the pending instructions in the speculation buffer β for its memory state, to be consistent with the speculation it might be in. The fetched instruction `atom` is timestamped using a timestamp greater than all the timestamps in both β and π. Finally, the fetch $(i, \texttt{atom})$ directive places the instruction in the pipelines. Here, no new cycle is necessary, hence $t = 0$, and the continuation s remains to be fetched and executed. The second rule, Cycle, is used when the state is not ready for `atom`. In that case, a cycle is executed, and the processor still has to fetch and execute $\texttt{atom}; s$.

Conditional The rules Spec-Cond-True-Correct and Spec-Cond-True-Incorrect define the behavior of the processor when encountering a conditional and the then-branch must be taken (i.e. when $b \neq 0$ in our language). The two rules presented can be decomposed into three steps: first the processor fetches the `jmp`; then executes it with the speculative execution of one of the branches; and finally, either continues normally the execution if the speculation was correct, or it rolls back if it mis-speculated.

The cost t is exactly the number of cycles needed to fetch the atomic jump (since the continuation is `skip`). Because the continuation is `skip`, no more rules can be applied, and the last rule applied is Atomic to fetch $\texttt{jmp}(b)$. Hence the jump is now in stage J_1, and we can consult the pipeline state to find which branch to take. We also obtain the timestamp k of the jump for the roll back.

In both rules, the predicted branch is then executed. The speculation lasts exactly $|\texttt{jmp}|$ cycles, which is checked by the Enforce-Cycle-* rules defined in Fig. 10: in case the branch and continuation are too short, we let the processor execute cycles on an empty program with judgment $(s,\omega) \xrightarrow{=}{}^{t} (s',\omega')$. After processing the jump, the history h is updated. The processor behavior after the speculation ends depends on the correctness of the prediction. If the processor correctly predicted the branch, then the continuation s' obtained after the speculation is used (rule Spec-Cond-True-Correct). Otherwise, the continuation and all instructions in π and β that were speculated are discarded (rule Spec-Cond-True-Incorrect). We keep the state σ_3 since committed instructions were necessarily older than the jump which was in J during the speculation. Finally, the processor restarts its execution from the correct branch s_1.

Remark that the history h does not change during the speculation. This is because the processor does not fetch another jump while there is already

$(p, \sigma, h) \Downarrow_t \sigma'$
executes the program p
from σ in t cycles

$$\textsc{Done}\quad \frac{(p; \texttt{skip}, \langle \sigma, \pi_\epsilon, h, \emptyset \rangle) \rightarrow^t (\texttt{skip}, \langle \sigma'', \pi, \_, \beta \rangle) \quad (\sigma'', \pi, \beta) \hookrightarrow^{t'} (\sigma', \pi_\epsilon, \emptyset)}{(p, \sigma, h) \Downarrow_{t+t'} \sigma'}$$

Fig. 11: Execution cost for small-step semantics

a jump in the pipeline. Therefore, two predictions cannot be interlaced: the branch history cannot change between the prediction of rule Spec-Cond-* and its update at the end of the rule.

Fetch and execution cost For any program p and processor state ω, the judgment $(p; \texttt{skip}, \omega) \rightarrow^t (\texttt{skip}, \omega')$ states that all instructions of p have been fetched in t cycles. If ω has empty an pipeline state π_ϵ and an empty speculation buffer, then t is the *fetch cost* of p. But not all instructions have been executed and committed after t cycles: some instructions may still be in π or β. To obtain the full execution cost, we need to keep executing cycles until we reach a pipeline state π_ϵ, where all the stages are empty (i.e. $\forall X_i, \pi_\epsilon(X_i) = \epsilon$), and an empty speculation buffer. This is captured by the judgment $(p, \sigma, h) \Downarrow_t \sigma'$, which gives the *execution cost* t of a program p starting with memory state σ and a branch predictor history h — see the Done rule in Fig. 11.

4 Static Analysis

We now present the static analysis technique we designed, which allows to obtain provable relational bounds of the execution cost of a program. To do this, we first instrument the original program s by adding a cost variable `cost`, such that the set of possible run-time values of `cost` in the instrumented program contains the exact value of the execution cost of s. We then perform a standard relational numerical static analysis on this instrumented program to obtain relational bounds between the original program cost and input variables (for instance the length of an input array). The instrumentation is performed using a standard may/must static analysis and a symbolic execution of instruction blocks.

The analysis algorithm is presented in Section 4.1, illustrated on an example and with the soundness theorem guaranteed. The soundness proof is detailed in Section 4.2.

4.1 Instrumentation for a numerical analysis

The instrumentation of each statement is defined by induction in Fig. 13 and the notations of the analyses are summarized in Fig. 12. For blocks — a sequence of atomic instructions $\texttt{atom}_1; \ldots; \texttt{atom}_n$ without control-flow structure — the instrumentation relies on a block cost approximations $[\![\mathsf{blk}]\!]^\sharp$ which outputs the

Alias analysis notations:

$\sigma^\sharp$	$\in \mathsf{S}_a^\sharp$	Abstract alias memory states
$[\![\texttt{atom}]\!]_{\mathrm{a}}^\sharp$	$\in \mathsf{S}_a^\sharp \to \mathsf{S}_a^\sharp$	Abstract alias semantics for an atomic instruction
$\bowtie_{\mathrm{May}}^\sharp, \bowtie_{\mathrm{Must}}^\sharp$	$\in \mathsf{Atoms} \times \mathsf{Atoms} \times \mathsf{S}_a^\sharp \to \mathsf{bool}$	No data-dependency test
$\iota_a^\sharp[s]$	$\in \mathsf{S}_a^\sharp$	Initial abstract alias memory state for the given statement s
γ_a	$\in \mathsf{S}_a^\sharp \to \mathcal{P}(\mathsf{S})$	Concretization function

Abstract states:

$\pi^\sharp$	$\in \mathsf{P}^\sharp = \mathsf{Stages} \to (\mathsf{Atoms} \cup \epsilon)$	Abstract pipeline state
$\pi_\epsilon^\sharp$	$\in \mathsf{P}^\sharp$	The empty abstract pipeline state

Numerical analysis notations:

$\sigma^\sharp$	$\in \mathsf{S}_n^\sharp$	Abstract numerical memory states
$[\![s]\!]_n^\sharp$	$\in \mathsf{S}_n^\sharp \to \mathsf{S}_n^\sharp$	Abstract numerical analysis of statement s
$\iota_n^\sharp[s]$	$\in \mathsf{S}_n^\sharp$	Initial abstract memory state for the given statement s
γ_n	$\in \mathsf{S}_n^\sharp \to \mathcal{P}(\mathsf{S} \times \mathsf{S})$	Concretization function returning pre and post states
proj_R	$\in \mathsf{S}_n^\sharp \to \mathsf{S}_n^\sharp$	Projects an invariant on registers R

Instrumentation notations:

$(\pi^\sharp, \sigma^\sharp, n)$	$\in \mathsf{I}^\sharp = \mathsf{P}^\sharp \times \mathsf{S}_a^\sharp \times \mathbb{N}$	Abstract processor state
$[\![s]\!]_{\bowtie^\sharp}$	$\in \mathsf{I}^\sharp \to \mathsf{I}^\sharp$	Abstract semantics of a statement s (parameterized by a no data-dependency test $\bowtie^\sharp$)
$\mathbb{T}$	$\in (\mathsf{Stmt} \times \mathsf{S}_a^\sharp) \to (\mathsf{Stmt} \times \mathsf{S}_a^\sharp)$	Instrumentation of a statement
$[\![\mathsf{blk}]\!]^\sharp$	$\in \mathsf{S}_a^\sharp \to (\mathbb{N} \times \mathbb{N} \times \mathsf{S}_a^\sharp)$	Cost analysis (lower and upper bounds) of a block with alias information

Fig. 12: Static analysis notation

bounds $[u, o]$ of the cost to execute blk. The instrumentation relies on an alias analysis — whose purpose is explained later — and is thus parameterized by an abstract memory state $\sigma^\sharp$ from the alias analysis. The instrumentation adds non-deterministic increment `cost += [u, o]` to the cost variable.

Instrumented programs are analyzed using a numerical analysis $[\![\cdot]\!]_n^\sharp$. We let R_0 be the input registers of our programs, and denote by $\iota_n^\sharp[s]$ the initial abstract memory state of the program s. Let s' be the instrumentation of a program s. To obtain the cost (invariant) $\mathbb{C}$ of s, we project the abstract numerical invariant of s' on the input registers R_0 and the cost variable:

$$\mathbb{C}(s) = \mathrm{proj}_{R_0 \cup \{\texttt{cost}\}}([\![s']\!]_n^\sharp(\iota_n^\sharp[s])) \quad \text{where} \quad (s', \_) = \mathbb{T}(s, \iota_a^\sharp[s]))$$

Block instrumentation The block instrumentation computes the cost with $[\![\mathsf{blk}]\!]^\sharp$. It performs two simulations $[\![\mathsf{blk}]\!]_{\bowtie_{\mathrm{Must}}^\sharp}$ and $[\![\mathsf{blk}]\!]_{\bowtie_{\mathrm{May}}^\sharp}$ of the block to obtain under and over approximations of the execution cost. To simulate the execution of a block, the analysis takes the instructions of the block in order and tries to

Block instrumentation:

$$[\![a]\!]_{\bowtie^\sharp}(\pi^\sharp,\sigma^\sharp,n) = \begin{cases} (\pi^\sharp[X_1 \mapsto a], [\![a]\!]^\sharp\sigma^\sharp, n) & \text{If } \exists X \in \min\{Y \in a \mid Y_1 = \epsilon\} \\ & \text{and } \forall a', \bowtie^\sharp (a, a', \sigma^\sharp) \text{ holds} \\ (\mathit{cycle}(\pi^\sharp), \sigma^\sharp, n+1) & \text{Otherwise} \end{cases}$$

$$[\![a_1; \ldots; a_n]\!]_{\bowtie^\sharp}\sigma^\sharp = [\![a_n]\!]_{\bowtie^\sharp} \circ \ldots \circ [\![a_1]\!]_{\bowtie^\sharp}(\pi^\sharp_\epsilon, \sigma^\sharp, 0)$$

$$[\![\mathsf{blk}]\!]^\sharp\sigma^\sharp_1 = (u, o + \max(\pi^\sharp), \sigma^\sharp_2) \quad \text{with} \quad \begin{array}{l} [\![\mathsf{blk}]\!]_{\bowtie^\sharp_{\text{Must}}}\sigma^\sharp_1 = (\_, \sigma^\sharp_2, u) \\ [\![\mathsf{blk}]\!]_{\bowtie^\sharp_{\text{May}}}\sigma^\sharp_1 = (\pi^\sharp, \_, o) \end{array}$$

Program instrumentation:

$$\mathbb{T}(\mathsf{blk}, \sigma^\sharp_1) = (\mathsf{blk};\ \texttt{cost += [u, o]}, \sigma^\sharp_2) \qquad \text{if } [\![\mathsf{blk}]\!]^\sharp\sigma^\sharp_1 = (u, o, \sigma^\sharp_2)$$

$$\mathbb{T}(s_1; s_2, \sigma^\sharp_1) = (s'_1; s'_2, \sigma^\sharp_3) \qquad \text{if } (s'_1, \sigma^\sharp_2) = \mathbb{T}(s_1, \sigma^\sharp_1) \text{ and } (s'_2, \sigma^\sharp_3) = \mathbb{T}(s_2, \sigma^\sharp_2)$$

If $(s'_1, \sigma^\sharp_2) = \mathbb{T}(s_1, [\![b]\!]^\sharp_a\sigma^\sharp_1)$ and $(s'_2, \sigma^\sharp_3) = \mathbb{T}(s_2, [\![\neg b]\!]^\sharp_a\sigma^\sharp_1)$:

$$\mathbb{T}(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma^\sharp_1) = (\texttt{cost += [0, L]}; \texttt{if } b \texttt{ then } s'_1 \texttt{ else } s'_2, \sigma^\sharp_2 \sqcup \sigma^\sharp_3)$$

If $\sigma^\sharp = \mathrm{lfp}(\lambda\Sigma \to \sigma^\sharp_0 \sqcup [\![s]\!]^\sharp_a \circ [\![b]\!]^\sharp_a\Sigma)$ and $\mathbb{T}(s, [\![b]\!]^\sharp_a\sigma^\sharp) = (s', \_)$:

$$\mathbb{T}(\texttt{while } b \texttt{ do } s \texttt{ done}, \sigma^\sharp_0) = (\texttt{while } b \texttt{ do } (\texttt{cost += [0, L]; } s') \texttt{ done; cost += [0, L]}, [\![\neg b]\!]^\sharp_a\sigma^\sharp)$$

Fig. 13: Instrumentation of a program ($L = |\texttt{jmp}|$)

fetch them. If no instruction can be fetched, e.g. because the first stage of all pipelines are full, or because of a data-dependency, it increments its cycle counter and updates its abstract pipeline state $\pi^\sharp$ with a function *cycle* — which makes instructions advance on stage forward in their pipelines. In these simulations, the pipeline abstract state $\pi^\sharp$ is a function from stages to unresolved instructions (the abstract simulation cannot resolve instructions, as this require a concrete memory state).

The simulation relies on an abstract memory state $\sigma^\sharp$ from an auxiliary alias analysis conducted in parallel to the instrumentation. This alias analysis is used to determine if there may be data-dependencies between the current instruction and any instruction in the pipelines, using an alias operator $\bowtie^\sharp$. The alias operator $\bowtie^\sharp$ used depends on how data-dependencies should be handled, which depends on whether we are computing the lower or upper-bound. When computing the lower bound, we are in the best-case scenario, and assume that there is a data-dependency — hence a delay — only if the memory location *must* always alias. Hence we require that the must-alias operator $\bowtie^\sharp_{\text{Must}}$ satisfies:

$$\neg \bowtie^\sharp_{\text{Must}} (\texttt{atom}, \texttt{atom}', \sigma^\sharp) \implies \forall \sigma \in \gamma(\sigma^\sharp), \mathrm{locks}(\texttt{atom}, \texttt{atom}', \sigma)$$

On the other hand, the upper bound corresponds to the worst-case scenario, and relies on a *may* alias analysis to detect instructions that may induce a delay: if an instruction is known never to alias with any instruction already in the pipeline,

no data-dependency delay needs to be added. We require that the may-alias operator $\bowtie^\sharp_{\text{May}}$ satisfies:

$$\bowtie^\sharp_{\text{May}}(\texttt{atom}, \texttt{atom}', \sigma^\sharp) \implies \forall \sigma \in \gamma(\sigma^\sharp), \neg\, \text{locks}(\texttt{atom}, \texttt{atom}', \sigma)$$

If there is no data-dependency, then the simulation finds an empty stage for `atom` and updates the alias analysis.

Example Consider the instrumentation of the program below. This program computes in register p the scalar product of two vectors stored in arrays A and B. We suppose that A and B do not alias at the beginning, and that the may and must alias analyses are able to determine that there is no aliasing between the address read l.14 and l.18. Each instruction is commented with the cycle at which it is fetched in its block, starting from an empty pipeline.

```
1  // Initialization
2  cost := 0;
3  p   := 0;    // 1
4  i   := 0;    // 1
5  r0  := n-i;  // 2
6  // Block's cost
7  cost += [1, 2] ;
8  while (r0 > 0) do
9    // Backtrack penalty
10   cost += [0, 4];
11   r1 := i*8;       // 1
12   a  := [A + r1]; // 6
17   r2 := i*8;       // 6
18   b  := [B + r2]; // 11
19   c  := a*b;       // 13
20   p  := p+c;       // 18
21   i  := i+1;       // 18
22   r0 := n-i;       // 19
23   // Block's cost
24   cost += [18, 19];
25 done;
26 // Backtrack penalty
27 cost += [0, 4];
```

Finally, we use a numerical static analysis to obtain the final value of the cost variable. On the example above, we assume that the inputs A and B are of size $n \geq 0$, and we select $R_0 = \{n\}$ as input register. Once projected, the relation between cost and the initial value of n gives a cost of the program in the interval $[1 + 18n; 6 + 23n]$.

The soundness of the static analysis is formalized in the following theorem where we used the concretization function γ_n to link the initial and final states.

Definition 1 (Initial states). *A memory state σ_0 is initial if it satisfies*

$$(\sigma_0, \sigma_0) \in \gamma_n(\iota^\sharp_n[s]) \ \wedge \ \sigma_0 \in \gamma_a(\iota^\sharp_a[s])$$

Theorem 1 (Static analysis soundness). *Let s be a program and σ_0 an initial state. Then, the computed numerical relation is a sound approximation of the execution cost of s from σ_0:*

$$\forall h, t, (s, \sigma_0, h) \Downarrow_t \_ \implies (\sigma_0, \{\mathsf{cost} \mapsto t\}) \in \gamma_n \circ \mathbb{C}(s)$$

4.2 Proof of soundness

To prove Theorem 1, we need to prove that: (i) the block approximation is sound; and (ii), the program instrumentation is sound.

The following theorem states the soundness of our block instrumentation.

$$\textsc{Block}\ \frac{s \text{ a block} \qquad [\![\mathsf{blk}]\!]^\sharp \sigma_1^\sharp = (u, o, {\sigma_2}^\sharp) \qquad \sigma_2 \in \mathbb{S}[\![s]\!]\sigma_1}{(s, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, {\sigma_2}^\sharp)}$$

$$\textsc{Seq-No-Block}\ \frac{s_1; s_2 \text{ not a block} \qquad (s_1, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, {\sigma_2}^\sharp) \qquad (s_2, \sigma_2, {\sigma_2}^\sharp) \Downarrow_{[u',o']} (\sigma_3, {\sigma_3}^\sharp)}{(s_1; s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u+u',o+o']} (\sigma_3, {\sigma_3}^\sharp)}$$

$$\textsc{Cond-True}\ \frac{[\![b]\!]\sigma_1 \neq 0 \qquad (\texttt{jmp}(b); s_1, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,\_]} (\sigma_2, {\sigma_2}^\sharp) \qquad (s_1, \sigma_1, \sigma_1^\sharp) \Downarrow_{[\_,o]} (\sigma_2, \sigma_2^\sharp)}{(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o+|\texttt{jmp}|]} (\sigma_2, {\sigma_2}^\sharp)}$$

$$\textsc{Cond-True}\ \frac{[\![b]\!]\sigma_1 = 0 \qquad (\texttt{jmp}(b); s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,\_]} (\sigma_2, {\sigma_2}^\sharp) \qquad (s_1, \sigma_2, \sigma_1^\sharp) \Downarrow_{[\_,o]} (\sigma_2, \sigma_2^\sharp)}{(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o+|\texttt{jmp}|]} (\sigma_2, {\sigma_2}^\sharp)}$$

$$\textsc{While}\ \frac{(\texttt{if } b \texttt{ then } (s; \texttt{while } b \texttt{ do } s \texttt{ done}), \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, {\sigma_2}^\sharp)}{(\texttt{while } b \texttt{ do } s \texttt{ done}, \sigma, \sigma^\sharp) \Downarrow_{[u,o]} (\sigma_2, {\sigma_2}^\sharp)}$$

Fig. 14: The big-step approximate semantics computes the cost bounds of statements, with the help of an alias abstract memory state $\sigma^\sharp$

Theorem 2 (Block approximation correction). *For any block* blk *and abstract memory state* $\sigma^\sharp$*:*

$$[\![\mathit{blk}]\!]^\sharp \sigma^\sharp = (u, o, \_) \ \Rightarrow\ \forall \sigma \in \gamma(\sigma^\sharp), t, h,\ ((\mathit{blk}, \sigma, h) \Downarrow_t \_ \Rightarrow t \in [u, o])$$

The theorem is proved by bi-simulation, by induction on the number of instructions of blk. For the lower bound, if the concrete semantics fetches an instruction, the correction of the must analysis ensures that the simulation will fetch it too. However, the abstract simulation of the pipeline state may fetch instruction earlier than the concrete semantics, e.g. when the must alias analysis does not detect that an aliasing always occurs. Thus the under-approximation cost is smaller or equal to the concrete cost.

For the upper bound, the converse reasoning applies. If the concrete semantics executes a cycle, because of a conflict, then the correction of the may alias analysis guarantees that the over-approximation also executes a cycle. The may analysis may not be able to statically prove that some instruction cannot alias with an instruction already in the pipeline, which can result in more cycles in the abstract semantics. Thus the over-approximation cost is larger or equal to the concrete cost.

Soundness of the program instrumentation We rely on an approximate program semantics to prove the soundness of our program instrumentation. This big-step semantics is defined inductively on the syntax, with a special case for blocks, and computes bounds for each statement. It abstracts away the reorder buffer

and the branch prediction history, keeping only the memory state σ and the abstract state $\sigma^\sharp$ computed by the alias analyses. Its rules are in Fig. 26 and follows the scheme of the instrumentation. It is straightforward to show that the cost-approximate semantics computes the same bounds than the ones of the cost variable in the instrumented program.

The cost-approximate semantics is sound w.r.t. the small-step semantics.

Theorem 3 (Cost-approximate soundness). *Let s be a program, σ_1 a memory state, $\sigma_1^\sharp$ an abstract alias state such that $\sigma_1 \in \gamma_a(\sigma_1^\sharp)$, and s' the instrumentation of s (i.e. $(s', \_) = \mathbb{T}(s, \sigma_1^\sharp)$), then*

$$\forall t, h, u, o, \sigma_2, \begin{pmatrix} (s, \sigma_1, h) \Downarrow_t \sigma_2 \\ \wedge\ (s, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, \_) \end{pmatrix} \implies \begin{pmatrix} \sigma_2[\mathsf{cost} \mapsto t] \in \mathbb{S}[\![s']\!]\sigma_1 \\ \wedge\ u \leq t \leq o \end{pmatrix}$$

Also, the existence of an execution in the small-step semantics is enough to guarantee the existence of bounds for the cost-approximate semantics.

Theorem 4 (Cost-approximate existence). *Let s be a program and σ_1 a memory state and $\sigma_1^\sharp$ an abstract alias state such that $\sigma_1 \in \gamma_a(\sigma_1^\sharp)$*

$$\forall t, h, \sigma_2,\ (s, \sigma_1, h) \Downarrow_t \sigma_2 \implies (\exists o, u, (s, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, \_))$$

For Theorem 3, only the second component of the conjunction requires a detailed proof — the other is a trivial property of the instrumentation. The proof of this theorem is given in Appendix G, and relies on several intermediate semantics, until we obtain a big-step semantics with immediate application of instructions on the memory state (i.e. where the effects of an instruction are applied immediately, and not when it is committed) and with approximations due to dropping the branch prediction history and concrete memory state in the block analysis.

Cost from a Non-Empty Pipeline State The difficulty of Theorem 3's proof is that the intermediate processor states in the small-step semantics do not necessarily have an empty pipeline state and empty speculation buffer, while Theorem 2 consider the execution cost of a block from an *empty pipeline state*.

Assume that we have two blocks blk_1 and blk_2 that are executed one after the other (e.g. blk_1 and blk_2 can be the body of a while loop). Then, blk_2 is executed starting from the processor state ω_1 resulting from blk_1's execution.

$$(\mathsf{blk}_1, \langle \sigma_1, \pi_\epsilon, h, \emptyset \rangle) \rightarrow^{t_1} \omega_1 \text{ and } (\mathsf{blk}_2, \omega_1) \rightarrow^{t_2} (\mathtt{skip}, \omega_2) \text{ and } \omega_2 \hookrightarrow^{t_2'} \langle \sigma', \pi_\epsilon, h', \emptyset \rangle$$

Here, we need to show that $t_1 + t_2 + t_2' \leq o_1 + o_2$, where:

$$(\mathsf{blk}_1, \sigma_1, \sigma_1^\sharp) \Downarrow_{[\_,o_1]} (\sigma_2, \sigma_2^\sharp) \quad \text{and} \quad (\mathsf{blk}_2, \sigma_2, \sigma_2^\sharp) \Downarrow_{[\_,o_2]} (\sigma', \sigma'^\sharp)$$

The fetch cost t_1 of blk_1 is smaller than its execution cost t_1'. Hence using Theorem 2:

$$(\mathsf{blk}_1, \sigma_1, h) \Downarrow_{t_1'} \sigma_2 \quad \text{and} \quad t_1 \leq t_1' \leq o_1$$

But we cannot bound the execution cost of blk_2 by o_2, because Theorem 2 only bounds the cost of executing blk_2 starting from an *empty pipeline and speculation buffer state*. Since it starts from a (potentially) non-empty state ω_1, t_2 may be strictly larger than o_2.

Intuitively, the cost approximation $t_1 + t_2 + t_2' \leq o_1 + o_2$ holds because the additional cost incurred when starting from an non-empty pipeline state has already been accounted by the *previous* block, i.e. in o_1. To formalize this, let $\max(\pi)$ be the maximum delay of all resources in π:

$$\max(\pi) = \max\Big(\underbrace{\max_{X_i \in \mathsf{Stages}, \pi(X_i) \neq \emptyset} (|\pi(X)| - i + 1)}_{\text{delays on locations}}, \underbrace{\max_{X \in \mathsf{Pips}} \mathbb{1}_{X_1 \neq \emptyset}}_{\text{delay for first stages}}\Big)$$

where $\mathbb{1}_C$ evaluates to 1 if the predicate C is true, 0 otherwise.

The following lemma guarantees that we do bound the cost of a statement by computing its cost from an empty pipeline.

Lemma 1. *Let $\langle \sigma, \pi, h, \beta \rangle$ be a processor state and s a program. Consider the following two executions starting from the pipeline and buffer states, resp., π, β and $\pi_\epsilon, \emptyset$.*

$$(s; \textbf{\textit{skip}}, \langle \sigma, \pi, h, \beta \rangle) \to^t (\textbf{\textit{skip}}, \langle \_, \pi', \_, \_ \rangle)$$
$$\textit{and } (s; \textbf{\textit{skip}}, \langle \sigma, \pi_\epsilon, h, \emptyset \rangle) \to^{t'} (\textbf{\textit{skip}}, \langle \_, \pi'', \_, \_ \rangle)$$

Then $t' \leq t$ and $t + \max(\pi') \leq \max(\pi) + t' + \max(\pi'')$

The proof, given in Appendix J.2, is not straightforward, and requires some care. Indeed, the two executions may not execute cycles synchronously: there is no guarantee that the execution which started with non-empty pipelines will execute a cycle when the other execution, which started from π_ϵ, does. To tackle this issue, we introduce the notion of *lateness*, a partial order relation on pipeline states that captures the fact that a pipeline state has already executed more cycles than another one. We prove that this partial ordering is preserved by our semantics.

Proof of Theorem 1 To conclude the proof of Theorem 1, let us take s a program, σ_0 an initial memory state, h a branch predictor history, such that the execution cost of s is t in the small-step semantics: $(s, \sigma_0, h) \Downarrow_t \sigma_1$. Recall that $\mathbb{C}(s) = \mathrm{proj}_{R_0 \cup \{\mathsf{cost}\}}([\![s']\!]_n^\sharp(\iota_n^\sharp[s]))$ with $\mathbb{T}(s, \iota_a^\sharp[s]) = (s', \_)$. By Theorem 4, there exists o and u such that $(s, \sigma_0, \sigma_0^\sharp) \Downarrow_{[u,o]} (\sigma_1, \_)$. By Theorem 3, $\sigma_1[\mathsf{cost} \mapsto t] \in \mathbb{S}[\![s']\!](\sigma_0)$.

Using the soundness of the numerical abstraction $[\![\cdot]\!]_n^\sharp$, we have

$$\forall \sigma^\sharp, \forall (\sigma_0, \sigma) \in \gamma_n(\sigma^\sharp),\ \{\sigma_0\} \times \mathbb{S}[\![s]\!]\sigma \subseteq \gamma_n([\![s]\!]_n^\sharp \sigma^\sharp)$$

and in particular $\{\sigma_0\} \times \mathbb{S}[\![s']\!]\sigma_0 \subseteq \gamma_n([\![s']\!]_n^\sharp \iota_n^\sharp[s])$. After projecting on R_0 and cost, we obtain $(\sigma_0, \{\mathsf{cost} \mapsto t\}) \in \gamma_n \circ \mathbb{C}(s)$ which concludes this proof.

5 Implementation

We implemented our instrumentation technique on top of Jasmin [2,3]. This framework allows to build high-assurance and high-speed cryptographic implementations by: i) combining low-level assembly instructions (e.g. flags and vectorized instructions) and high-level structured control flow; ii) using a verified compiler (à la CompCert [24]), with a mechanized Coq proof of behavior preservation; iii) verification tools for proving properties of Jasmin programs, including an embedding of Jasmin in the Easycrypt proof assistant [5], and a static analyzer to check the memory safety of Jasmin programs. The Jasmin compiler performs several compilation passes, such as dead-code elimination, function call inlining, and sharing of stack variables. All these compilation passes are proven correct in Coq (i.e. they preserve the semantics of programs)[5].

We have integrated our cost analysis late enough in the compilation chain in order to avoid change of the cost between the intermediate representation that is analyzed and the final assembly code that is generated by the compiler. Our analysis is implemented in OCaml and currently not verified in Coq. The analysis is parameterized by a user-given processor specification file, listing the instructions, their latency and the pipelines supporting them.

By default, the instrumentation respects the approximation semantics by making no assumption on the branch predictor. In the worst-case scenario the instrumentation thus considers that the branching always mis-predicts. We also provide an option that lets the user assume a basic branch predictor for the processor, which always tries to take the same branch as previously taken. Such a branch predictor can only mis-predict twice on a given while loop execution: when it enters and when it leaves.

The alias and numerical static analyzer (mentioned in Section 4) have been obtained by modifying the Jasmin static analyzer. This analyzer, which uses abstract interpretation techniques [12], was initially introduced in [3] to prove safety, and was executed before any compilation pass. Our cost analysis is run later in the compilation chain and it has been necessary to enhance the Jasmin relational numerical analysis with a *dynamic packing* technique, which handles the same variable with different degrees of precision at different program points. This a slight variation of the *packing* technique introduced in [13] where packs of variable where fixed at the level of block/function.

6 Experiments

We evaluate our cost analysis on different implementations of cryptographic primitives written in Jasmin. Examples include Poly1305 [7], a lookup-table-based implementation of AES [15], ChaCha20 [9] and multiplication in the finite field $\mathbb{F}_p$ with $p = 2^{255} - 19$. The latter is a core routine of the Curve25519 key exchange [8]. We report our experiments in Fig. 15. For some examples we

[5] Currently, Jasmin only supports x86 architectures. Note however that our method is not specific to x86, and can be applied to other architectures.

Programs	Lower bound	Upper bound	Naive upper bound
scalar prod (ref)	44 len	44 len + 8	46 len + 11
scalar prod (opt)	17.5 len - 23.5	17.5 len + 33	20 len + 39
poly1305 (ref)	7 len + 25	7.1 len + 150	7.5 len + 177
poly1305 (opt)	2.1 len + 25	2.2 len + 1410	3.9 len + 1098
aes	44.8 len + 446	44.9 len + 1115	50.7 len + 1946
chacha (ref)	16.2 len + 23	16.4 len + 1052	17.6 len + 1040
chacha (opt)	4 len + 27	4.1 len + 2130	5.7 len + 3035
fe25519_mul	427	427	464

Fig. 15: Experimental results.

report results for both a reference ("ref") and a hand-optimized ("opt") implementation. When cost depends on the (length of) inputs, our tool computes a symbolic cost w.r.t. to a variable len; for AES and ChaCha encryption and Poly1305 authentication this variable is the length of the input message. In the invariant computed by the numerical analysis, we only keep the best asymptotic constraint when several bounds were available. The tests were done assuming a basic branch predictor. The only target architecture currently supported by Jasmin is AMD64 (also known as x86-64 or x64). There are only very few in-order AMD64 CPUs; for our experiments we decided to approximate one of them, namely the Intel Atom 330. The pipeline structure and instruction latencies are modeled according to the documentation in Fog's CPU manuals [17,18].

We compare our results with a reference naive analysis (last column in Fig. 15) that over-approximates the cost of any block of atomic instructions by the sum of the latencies of each instruction. This approach hence coincides with state-of-the-art cost analyzer that do not take into account instruction pipelining. We also compare the reference programs to their hand-optimized variant, if available. For all programs we obtain a smaller upper-bound than the naive analysis. It shows that our bound computation is likely to improve precision over cost analyzers that ignore instruction pipelining. Our lower and upper-bounds are asymptotically very close, which shows that our cost analysis is asymptotically precise. For programs with hand-optimized version, the upper bound of the optimized program is asymptotically smaller than the lower bound of the original program. This shows our tool usefulness in proving the impact of programmer optimizations.

7 Related work

Starting from the seminal work of Wegbreit [29], there has been a large body of work for analyzing the cost of programs using recurrence relations [1], program logics [26], type systems [27,21,14,23], and static analysis [19]. These approaches rely on sophisticated methods for computing numerical invariants and inferring iterations bounds for loops or recursive computations. Our method allows to

leverage these powerful methods in a more realistic cost model that accommodates cost-critical micro-architectural features.

Cost analysis is also useful for reasoning about side-channel leakage. Ngo *et al* [25] define the constant-resource policy, an observational information flow policy which guarantees that the execution cost of a program does not depend on its secret inputs. Their analysis is an instance of a relational cost analysis [11], a variant of cost analysis that computes lower and upper bounds for the relative cost of two programs. These works are carried in the setting of a simple cost model; applying our cost model and methodology to side-channel analysis is an interesting direction for future work.

An alternative is to carry dynamic analyses with cycle-accurate cost models. For instance, Yourst [31] develops a model for a x86-64 processor. Dynamic approaches trade off precision for generality — bounds are for specific inputs. However, it would be interesting to explore if cycle-accurate cost models could be used for refining instrumentation.

An even simpler approach is to measure execution time for a large number of inputs. When combined with a statistical analysis, this approach yields a useful heuristic for analyzing if cryptographic implementations leak [28]. However, this approach does not provide any guarantee.

Worst Case Execution Time (WCET) analysis is a well-known industrial success in cost analysis. Using Abstract Interpretation, state-of-the-art analyzers are able to predict a safe upper-bound for embedded micro-architectures with strict real-time constraints. They take into account several advanced architectural optimizations, including pipelines and caches [16,30,20]. Our approach differs in scope, precision and semantic foundations. We focus our reasoning on instruction scheduling and provide feedback to programmer who want to hand-optimize their program, like in cryptographic implementation. Our abstraction is more coarse (e.g., we do not try to merge symbolic pipelines on junction points), but already precise enough for the cryptographic application area. WCET tools are clearly more ambitious in term of cost model and precision but they do not ground their work on a semantic model with the same level of mathematical rigour than us. We consider our work as an attempt to reconcile cost precision and rigorous semantic proofs. We also believe that our instrumentation approach can be more easily connected to previous foundational cost analysis works [22] by reusing off-the-shelf cost analyzers.

8 Conclusion

We developed a precise cost semantics for pipelined-optimized softwares executed on in-order processors. The semantics is suitable for automatic cost analysis and formal semantic proofs of soundness. Preliminary experiments demonstrate that our automatic analysis is more accurate than a naive cost analysis.

One direction for future work would be to extend our cost semantics with a cache model and extend our analysis with a may/must tracking of cache misses.

An other perspective is to formalize in Coq the soundness of our cost analysis in order to integrate it with the Jasmin high-assurance Coq framework.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. J. Autom. Reason. **46**, 161–203 (2011)
2. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.: Jasmin: High-assurance and high-speed cryptography. In: Proc. of CCS'2017. pp. 1807–1823. ACM (2017)
3. Almeida, J.B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., Strub, P.: The last mile: High-assurance and high-speed cryptographic implementations. In: In Proc of S&P'2020. pp. 965–982. IEEE (2020)
4. Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B.: SoK: Computer-aided cryptography. IACR Cryptol. ePrint Arch. p. 1393 (2019)
5. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.Y.: EasyCrypt: A tutorial. In: Foundations of Security Analysis and Design VII. pp. 146–166. Springer (2013)
6. Barthe, G., Koutsos, A., Mirliaz, S., Pichardie, D., Schwabe, P.: Semantic foundations for cost analysis of pipeline-optimized programs (2022), https://hal.inria.fr/hal-03779257, companion report
7. Bernstein, D.J.: The Poly1305-AES message-authentication code. In: Proc. of FSE'2005. LNCS, vol. 3557, pp. 32–49. Springer (2005)
8. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Proc of PKC'2006. LNCS, vol. 3958, pp. 207–228. Springer-Verlag (2006)
9. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: Workshop Record of SASC 2008: The State of the Art of Stream Ciphers (2008)
10. Cauligi, S., Disselkoen, C., Gleissenthall, K.v., Tullsen, D., Stefan, D., Rezk, T., Barthe, G.: Constant-time foundations for the new spectre era. In: Proc. of PLDI'2020. p. 913–926. ACM (2020)
11. Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational cost analysis. In: Proc. of POPL'17. pp. 316–329. ACM (2017)
12. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77. pp. 238–252. ACM (1977)
13. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Proc. of ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer (2005)
14. Crary, K., Weirich, S.: Resource bound certification. In: Proc. of POPL'00. pp. 184–198. ACM (2000)
15. Daemen, J., Rijmen, V.: AES proposal: Rijndael, version 2 (1999), http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf
16. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Proc. of EMSOFT'01. LNCS, vol. 2211, pp. 469–485. Springer (2001)
17. Fog, A.: Instruction tables (2020), https://www.agner.org/optimize/instruction_tables.pdf

18. Fog, A.: The microarchitecture of Intel, AMD and VIA CPUs – An optimization guide for assembly programmers and compiler makers (2020), https://www.agner.org/optimize/microarchitecture.pdf
19. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Proc. of POPL'09. pp. 127–139. ACM (2009)
20. Hahn, S., Reineke, J.: Design and analysis of SIC: A provably timing-predictable pipelined processor core. In: Proc. of RTSS'18. pp. 469–481. IEEE Computer Society (2018)
21. Hughes, J., Pareto, L.: Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In: Proc. of ICFP'99. pp. 70–81. ACM (1999)
22. Knoth, T., Wang, D., Polikarpova, N., Hoffmann, J.: Resource-guided program synthesis. In: Proc. of PLDI'19. pp. 253–268. ACM (2019)
23. Knoth, T., Wang, D., Reynolds, A., Hoffmann, J., Polikarpova, N.: Liquid resource types. Proc. of ICFP'20 pp. 106:1–106:29 (2020)
24. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4), 363–446 (2009)
25. Ngo, V.C., Dehesa-Azuara, M., Fredrikson, M., Hoffmann, J.: Verifying and synthesizing constant-resource implementations with types. In: Proc.of SP'17. pp. 710–728. IEEE Computer Society (2017)
26. Nielson, H.R.: A Hoare-like proof system for analysing the computation time of programs. Sci. Comput. Program. **9**(2), 107–136 (1987)
27. Reistad, B., Gifford, D.K.: Static dependent costs for estimating execution time. In: Proc. of ACM Conference on LISP and Functional Programming. pp. 65–78. ACM (1994)
28. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Proc. of DATE'17. pp. 1697–1702. IEEE (2017)
29. Wegbreit, B.: Verifying program performance. J. ACM **23**(4), 691–699 (1976)
30. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **28**(7), 966–978 (2009)
31. Yourst, M.T.: Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In: Proc. of ISPASS'19. pp. 23–34. IEEE Computer Society (2007)

A Auxiliary functions

Read and write The precise definition of the read and write functions on atomic instruction are displayed below. To obtain the registers from operands we use an operator op.

$$\begin{aligned} \mathrm{op}(r) &= \{r\} && \text{if } r \in \mathsf{Reg} \\ \mathrm{op}(n) &= \emptyset && \text{otherwise, } n \in \mathsf{Val} \end{aligned}$$

For any $\bowtie \in \{+, -, \leq, \times\}$,

$$\begin{aligned}
\mathrm{read}(r := o_1 \bowtie o_2, \sigma) &= \mathrm{op}(o_1) \cup \mathrm{op}(o_2) \\
\mathrm{read}(r_1 := [r_2 + o], \sigma) &= \{r_2, \sigma(r_2) + \sigma(o)\} \cup \mathrm{op}(o) \\
\mathrm{read}([r + o_1] := o_2), \sigma) &= \{r\} \cup \mathrm{op}(o_1) \cup \mathrm{op}(o_2) \\
\mathrm{read}(\mathtt{jmp}(o), \sigma) &= \mathrm{op}(o)
\end{aligned}$$

$$\begin{aligned}
\mathrm{write}(r := _, \sigma) &= \{r\} \\
\mathrm{write}([r + o] := _, \sigma) &= \{\sigma(r) + \sigma(o)\} \\
\mathrm{write}(\mathtt{jmp}(o), \sigma) &= \emptyset
\end{aligned}$$

Instruction resolution The resolution of an instruction consist in replacing all registers read by their value in σ. For brevity, we adopt the convention that $\sigma(n) = n$ where $n \in \mathsf{Val}$ is a constant. This avoid distinct cases on the operators.

For any $\bowtie \in \{+, -, \leq, \times\}$,

$$\begin{aligned}
\mathrm{resolve}(r := o_1 \bowtie o_2, \sigma) &= (r := \sigma(o_1) \bowtie \sigma(o_2)) \\
\mathrm{resolve}(r_1 := [r_2 + o], \sigma) &= (r_1 := [\sigma(r_2) + \sigma(o)]) \\
\mathrm{resolve}([r + o_1] := o_2, \sigma) &= ([\sigma(r) + \sigma(o_1)] := \sigma(o_2)) \\
\mathrm{resolve}(\mathtt{jmp}(o), \sigma) &= (\mathtt{jmp}(\sigma(o)))
\end{aligned}$$

The read and write functions are extended to transient instruction resulting from resolve.

$$\begin{aligned}
\mathrm{read}(r := [l + n]) &= \{l + n\} \\
\mathrm{read}(_) &= \emptyset
\end{aligned}
\qquad
\begin{aligned}
\mathrm{write}(r := _) &= \{r\} \\
\mathrm{write}([l + n] := _) &= \{l + n\} \\
\mathrm{write}(\mathtt{jmp}(o)) &= \emptyset
\end{aligned}$$

B Small-step semantics with explicit speculation buffer

The section gives the complete rules of small-step semantics. In particular it displays all the rules for skip, sequence and while loop in Fig. 16

C Proof of approximation soundness

The soundness of the cost-approximate semantics, Theorem 3, is proven gradually through two intermediate big-step semantics. Their goal is to abstract away details of the processor state while keeping the same cost.

Fig. 17 summarizes the approach. The appendixes present the two intermediate semantics, namely the Concrete Big-Step Pipeline Semantics in Appendix D and the Immediate Big-Step Pipeline Semantics in Appendix E, and then prove

$(s, \omega) \to^t (s', \omega')$
execute t cycles and fetch as much instructions of s as possible before each cycle

Atomic
$$\frac{i = \max(\beta, \pi) + 1 \qquad \mathrm{ready}(\texttt{atom}, \beta(\sigma), \pi) \qquad (\beta(\sigma), \pi) \xhookrightarrow[\text{fetch } (i, \texttt{atom})]{} \pi'}{(\texttt{atom}; s, \langle \sigma, \pi, h, \beta \rangle) \to^0 (s, \langle \sigma, \pi', h, \beta \rangle)}$$

Cycle
$$\frac{\neg\, \mathrm{ready}(\texttt{atom}, \beta(\sigma), \pi) \qquad (\sigma, \pi, \beta) \hookrightarrow (\sigma', \pi', \beta')}{(\texttt{atom}; s, \langle \sigma, \pi, h, \beta \rangle) \to^1 (\texttt{atom}; s, \langle \sigma', \pi', h, \beta' \rangle)}$$

Steps
$$\frac{(s, \omega) \to^t (s'', \omega'') \qquad (s'', \omega'') \to^{t'} (s', \omega')}{(s, \omega) \to^{t+t'} (s', \omega')}$$

Seq
$$\frac{(s_1; (s_2; s_3), \omega) \to^t (s', \omega')}{((s_1; s_2); s_3, \omega) \to^t (s', \omega')}$$

Skip
$$\frac{}{(\texttt{skip}; s, \omega) \to^0 (s, \omega)}$$

Enforce-Cycles
$$\frac{(s, \omega) \to^t (s', \omega')}{(s, \omega) \xrightarrow{=}{}^t (s', \omega')}$$

Enforce-Cycle-Exact
$$\frac{(s, \omega) \to^k (\texttt{skip}, \omega'') \qquad \omega'' \hookrightarrow^{t-k} \omega'}{(s, \omega) \xrightarrow{=}{}^t (\texttt{skip}, \omega')}$$

Spec-Cond-True-Correct
$$\frac{\begin{array}{c}(\texttt{jmp}(b); \texttt{skip}, \omega) \to^t (\texttt{skip}, \langle \sigma_2, \pi_2, h, \beta_2 \rangle) \qquad \pi_2(J_1) = (k, \texttt{jmp} : v) \\ v \neq 0 \qquad \neg\, \text{BP-predict}(\ell, h) \qquad h' = \text{BP-update}(\ell, h, \mathit{false}) \\ (s_1; s_3, \langle \sigma_2, \pi_2, h, \beta_2 \rangle) \xrightarrow{=}{}^{|\texttt{jmp}|} (s', \langle \sigma_3, \pi_3, h, \beta_3 \rangle)\end{array}}{(\ell : \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2; s_3, \omega) \to^{t+|\texttt{jmp}|} (s', \langle \sigma_3, \pi_3, h', \beta_3 \rangle)}$$

Spec-Cond-True-Incorrect-Deplete
$$\frac{\begin{array}{c}(\texttt{jmp}(b); \texttt{skip}, \omega) \to^t (\texttt{skip}, \langle \sigma_2, \pi_2, h, \beta_2 \rangle) \qquad \pi_2(J_1) = (k, \texttt{jmp} : v) \\ v \neq 0 \qquad \text{BP-predict}(\ell, h) \qquad h' = \text{BP-update}(\ell, h, \mathit{false}) \\ (s_2; s_3, \langle \sigma_2, \pi_2, h, \beta_2 \rangle) \xrightarrow{=}{}^{|\texttt{jmp}|} (\_, \langle \sigma_3, \pi_3, h, \beta_3 \rangle)\end{array}}{(\ell : \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2; s_3, \omega) \to^{t+|\texttt{jmp}|} (s_1; s_3, \langle \sigma_3, \pi_3[j : j \leq k], h', \beta_3[j : j \leq k] \rangle)}$$

While
$$\frac{(\ell : \texttt{if } b \texttt{ then } (s; \ell : \texttt{while } b \texttt{ do } s \texttt{ done}), \omega) \to^t (s', \omega')}{(\ell : \texttt{while } b \texttt{ do } s \texttt{ done}, \omega) \to^t (s', \omega')}$$

Done
$$\frac{(p; \texttt{skip}, \langle \sigma, \pi_\epsilon, h, \emptyset \rangle) \to^t (\texttt{skip}, \langle \sigma'', \pi, \_, \beta \rangle) \qquad (\sigma'', \pi, \beta) \hookrightarrow^{t'} (\sigma', \pi_\epsilon, \emptyset)}{(p, \sigma, h) \Downarrow_{t+t'} \sigma'}$$

Fig. 16: Small-step semantics with explicit speculation

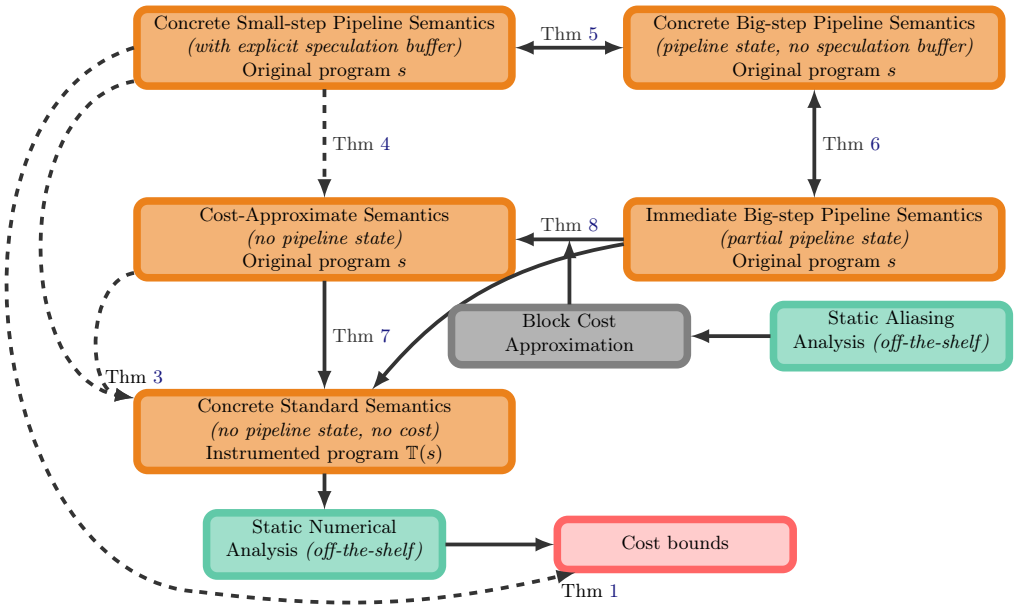


Fig. 17: Summary of our approach.

the three equivalence theorems. First, the equivalence of the two concrete semantics with Theorem 5, then the equivalence of the two intermediate big-step semantics with Theorem 6 and finally the equivalence of the immediate semantics with the cost-approximate semantics with Theorem 7. Together they form the proof of Theorem 3.

D Concrete big-step semantics without speculation buffer

This section presents an intermediate semantics, between the small-step introduced in Section 3 and the immediate big-step semantics introduce later in Appendix E. This semantics tracks the state of the pipeline stages, similarly to the small-step. But it follows a big-step scheme, like the cost-approximate semantics and drops the speculation buffer. The speculation buffer has in fact no impact on the cost. Also, our omniscient semantics does not need to wait the end of the jump processing to know which branch to execute, they only need to take into account the *cost* of the potential mispresdiction, not its modification on the processor that will roll backed anyway.

In this new semantics the pipeline state π is simplified since timestamps of the instructions are no longer required. As for the processor state ω, it no longer needs the speculation buffer β. This new processor notation is summarized in Fig. 18.

Processor state:

$\pi \in \mathsf{Stages} \to (\mathsf{Atoms_t} \cup \epsilon)$	Pipeline state
$\omega = \langle \sigma, \pi, h \rangle$	Processor state

Fig. 18: Processor state in the concrete big-step semantics

$$\textsc{Fetch}\ \frac{X = \min\{Y \in \texttt{atom} \mid \pi(Y_1) = \varepsilon\} \quad \pi' = \pi[X_1 \mapsto \mathrm{resolve}(\texttt{atom}, \sigma)]}{(\sigma, \pi) \xhookrightarrow[\mathsf{fetch}\ \texttt{atom}]{CB} \pi'} \qquad \textsc{Ready}\ \frac{\neg\, \mathrm{locks}(\texttt{atom}, \sigma, \pi) \quad X \in \texttt{atom} \quad \pi(X_1) = \varepsilon}{\mathrm{ready}(\texttt{atom}, \sigma, \pi)}$$

$$\mathrm{retired}'(\pi) = \{\pi(X_i) \mid \exists X_i \in \mathsf{Stages}, |\pi(X_i)| = i\}$$

$$\textsc{One-Cycle}\ \frac{\pi' = \pi[\forall X_i, X_i \mapsto \mathrm{next}(\pi, X_i)] \quad \sigma' = \bigcirc_{\texttt{atom} \in \mathrm{retired}'(\pi)} \mathbb{S}[\![\texttt{atom}]\!](\sigma)}{(\sigma, \pi) \hookrightarrow (\sigma', \pi')}$$

$$\textsc{Cycle-Release}\ \frac{t > 0 \quad (\sigma, \pi) \hookrightarrow^{t-1} (\sigma'', \pi'') \quad (\sigma'', \pi'') \hookrightarrow (\sigma', \pi') \quad \neg\, \mathrm{ready}(\texttt{atom}, \sigma'', \pi'') \quad \mathrm{ready}(\texttt{atom}, \sigma', \pi')}{(\sigma, \pi) \xhookrightarrow[\mathsf{cycle}\ \texttt{atom}]{CB\ \ t} (\sigma', \pi')}$$

$$\textsc{No-Cycle-Release}\ \frac{\mathrm{ready}(\texttt{atom}, \sigma, \pi)}{(\sigma, \pi) \xhookrightarrow[\mathsf{cycle}\ \texttt{atom}]{CB\ \ 0} (\sigma, \pi)}$$

Fig. 19: Directives of the concrete big-step semantics

Directives The new semantics uses three directives: fetch `atom`, cycle and a new directive cycle `atom` which executes just enough cycles so that `atom` can be fetched. The rules defining these directives are given in Fig. 19. The rule for fetch is similar to the small-step semantics, the timestamp of the instruction has just been dropped. The statement ready is also similar to the small-step semantics, checking the availability of a pipeline and the absence of data-dependency. Applying a cycle has the combined effect of the small-step cycle and commit: instructions progress on their pipeline and when they retire, according to retired, they are committed on the memory state σ. The order of this application is irrelevant since there are no data-dependency nor control-dependency.

Executing cycle for an atomic instruction, directive cycle `atom` is defined incrementally. The processor will execute just enough cycle for its state to be ready to fetch `atom`.

Big-step Since we no longer need to represent speculative executions, we drop the continuation style and simply use a big-step semantics. This big-step semantics computes the cost of a whole program by relying on the directives to get the cost of atomic instructions. Similarly to the small-step semantics, we defined both the *fetch cost* and the *execution cost* of a program. Some selected semantics rules are given in Fig. 20, and described below.

$$\textsc{Atomic}\ \frac{(\sigma,\pi) \xhookrightarrow[\text{cycle } \texttt{atom}]{CB\quad t} (\sigma',\pi'') \qquad (\sigma',\pi'') \xhookrightarrow[\text{fetch } \texttt{atom}]{CB} \pi'}{(\texttt{atom},\langle\sigma,\pi,h\rangle)\ {}^{CB}\Downarrow^{t}\ \langle\sigma',\pi',h\rangle} \qquad \textsc{Skip}\ \frac{}{(\texttt{skip},\omega)\ {}^{CB}\Downarrow^{0}\ \omega} \qquad \textsc{Seq}\ \frac{(s_1,\omega)\ {}^{CB}\Downarrow^{t}\ \omega'' \qquad (s_2,\omega'')\ {}^{CB}\Downarrow^{t'}\ \omega'}{(s_1;s_2,\omega)\ {}^{CB}\Downarrow^{t+t'}\ \omega'}$$

$$\textsc{Cond-True-Correct}\ \frac{(\texttt{jmp}(b),\omega)\ {}^{CB}\Downarrow^{t}\ \langle\sigma,\pi,h\rangle \qquad \sigma(b)\neq 0 \qquad \neg\,\text{BP-predict}(\ell,h) \qquad h' = \text{BP-update}(\ell,h,false) \qquad (s_1,\langle\sigma,\pi,h'\rangle)\ {}^{CB}\Downarrow^{t'}\ \omega'}{(\ell : \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\omega)\ {}^{CB}\Downarrow^{t+t'}\ \omega'}$$

$$\textsc{Cond-True-Incorrect}\ \frac{(\texttt{jmp}(b),\omega)\ {}^{CB}\Downarrow^{t}\ \langle\sigma,\pi,h\rangle \qquad \sigma(b)\neq 0 \qquad \text{BP-predict}(\ell,h) \qquad h' = \text{BP-update}(\ell,h,false) \qquad (\sigma,\pi)\hookrightarrow^{|\texttt{jmp}|}(\sigma',\pi') \qquad (s_1,\langle\sigma',\pi',h'\rangle)\ {}^{CB}\Downarrow^{t'}\ \omega'}{(\ell : \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\omega)\ {}^{CB}\Downarrow^{t+|\texttt{jmp}|+t'}\ \omega'}$$

$$\textsc{While}\ \frac{(\ell : \texttt{if}\ b\ \texttt{then}\ (s;\ell : \texttt{while}\ b\ \texttt{do}\ s\ \texttt{done}),\omega)\ {}^{CB}\Downarrow^{t}\ \omega'}{(\ell : \texttt{while}\ b\ \texttt{do}\ s\ \texttt{done},\omega)\ {}^{CB}\Downarrow^{t}\ \omega'}$$

$$\textsc{Done}\ \frac{(s,\langle\sigma,\pi_\epsilon,h\rangle)\ {}^{CB}\Downarrow^{t}\ (\sigma'',\pi,\_) \qquad (\sigma'',\pi)\hookrightarrow^{t'}(\sigma',\pi_\epsilon)}{(s,\sigma,h)\ {}^{CB}\Downarrow_{t+t'}\ \sigma'\checkmark} \quad \text{with } \forall X_i, \pi_\epsilon(X_i) = \epsilon$$

Fig. 20: Concrete big-step semantics of a multi-pipelined processor.

Fetch cost The judgment $(s,\omega)\ {}^{CB}\Downarrow^{t}\omega'$ states that, starting from the processor state ω, we fetch the program s in t cycles, and end in the processor state ω'.

Atomic Rule Atomic states that the fetch cost of an atomic instruction is the number of cycles necessary for the processor state to be ready for `atom`. This amount of cycles is obtained and applied through the cycle `atom` directive. Then, the directive fetch is called to update the pipeline state π.

Conditional We display in Fig. 20 the Cond-True-Correct and Cond-True-Incorrect rules, which handle conditionals when the predicate b holds. The rules premises are similar to their small-step semantics counter-parts. First, we need t cycles to fetch the `jmp`. This yields a state σ, from which we obtain the value of the register b. In the case of a correct prediction, the branch s_1 starts its execution. Once the prediction is confirmed, $|\texttt{jmp}|$ cycles later, the history is updated. As in the small-step semantics, the history remained unchanged during the speculation. Since the state (σ,π) is not modified once the prediction is confirmed, it is as if we simply fetched s_1 on $\langle\sigma,\pi,h'\rangle$. This yields the premise of rule Cond-True-Correct. In the case of a misprediction, $|\texttt{jmp}|$ cycles are

Delays:

$w \in \mathsf{Location} \to \mathbb{N}$	Writing delays
$r \in \mathsf{MemLocs} \to \mathbb{N}$	Reading delays
$p \in \mathsf{Pips} \to \{0,1\}$	Pipelines delays
$j \in \mathbb{N}$	Jump delay

Pipeline state:

$\pi ::= \langle w, r, p, j\rangle$	Pipeline state
$\omega ::= \langle \sigma, \pi, h\rangle$	Processor state

Fig. 21: Immediate pipelined processor

$$\textsc{Write Update}\ \frac{W = \mathrm{write} \circ \mathrm{resolve}(\mathtt{atom}, \sigma)}{(\mathtt{atom}, \sigma, w) \xhookrightarrow[write]{} w[W \mapsto |\mathtt{atom}|]} \qquad \textsc{Read Update}\ \frac{R = \mathrm{read} \circ \mathrm{resolve}(\mathtt{atom}, \sigma) \quad r' = r[\forall x \in R, x \mapsto \max(r(x), |\mathtt{atom}|)]}{(\mathtt{atom}, \sigma, r) \xhookrightarrow[read]{} r'}$$

$$\textsc{Pipeline Update}\ \frac{X = \min\{Y \in \mathtt{atom} \mid p(Y) = 0\}}{(\mathtt{atom}, p) \xhookrightarrow[pipeline]{} p[X \mapsto 1]} \qquad \textsc{Fetch}\ \frac{(\mathtt{atom}, \sigma, w) \xhookrightarrow[write]{} w' \quad (\mathtt{atom}, \sigma, r) \xhookrightarrow[read]{} r' \quad (\mathtt{atom}, p) \xhookrightarrow[pipeline]{} p' \quad \mathtt{atom} \neq \mathtt{jmp}(\_)}{(\sigma, \langle w, r, p, j\rangle) \xhookrightarrow[\mathsf{fetch}\ \mathtt{atom}]{} (\mathbb{S}[\![\mathtt{atom}]\!]\sigma, \langle w', r', p', j\rangle)}$$

$$\textsc{Fetch Jump}\ \frac{}{(\sigma, \langle w, r, p, j\rangle) \xhookrightarrow[\mathsf{fetch}\ \mathtt{jmp}(b)]{} (\sigma, \langle w, r, p[J \mapsto 1], |\mathtt{jmp}|\rangle)}$$

Fig. 22: Directives of the immediate semantics

executed, but once the prediction is invalidated all that was fetched and retired is roll backed. From a external point of view, it is as if we simply executed $|\mathtt{jmp}|$ cycles, without fetching anything. This is the premise of rule Cond-True-Incorrect, which then start the fetching of the correct branch.

E Immediate Big-step Pipeline semantics

The concrete big-step semantics is still more complex than needed for cost-computation. For instance it needs to keep the content of each stage to apply their effect on the memory state σ when they retire. But since the processor must respect the sequential semantics, and since it ensures that by delaying instruction when there is a data-dependency, then there is no need to delay the application of instruction on the memory state. And since we do not need to delay it, we do not need to store exactly the instruction in each stage. In conclusion, a simpler semantics can only reason on the *resources* (locations or pipelines) to compute the cost of a program. This simpler semantics is the immediate big-step semantics, detailed in this section.

Pipeline state A pipeline state comprises four components $\langle w, r, p, j\rangle$ (see Fig. 21). The delays – in cycles – until a variable is available in writing and

$$\textsc{Cycle}\ \frac{t > 0 \qquad w' = \lambda x.\max(0, w(x) - t) \qquad r' = \lambda x.\max(0, r(x) - t) \qquad p' = \lambda X.0 \qquad j' = \max(0, j - t)}{\langle w, r, p, j\rangle \hookrightarrow\!\!\!\!\rightarrow^{t} \langle w', r', p', j'\rangle}$$

$$\textsc{Cycle-Release-Jump}\ \frac{t = \max\left(p(J), j, w(b)\right) \qquad \langle w, r, p, j\rangle \hookrightarrow\!\!\!\!\rightarrow^{t} \pi'}{(\sigma, \langle w, r, p, j\rangle) \xhookrightarrow[\mathsf{cycle}\ \mathtt{jmp}(b)]{t} \pi'}$$

$$\textsc{Cycle-Release}\ \frac{\mathtt{atom} \neq \mathtt{jmp}(\_) \qquad W = \mathrm{write}(\mathtt{atom}, \sigma) \qquad R = \mathrm{read}(\mathtt{atom}, \sigma) \qquad t = \max\left(\min_{X \in \mathtt{atom}} p(X), \max_{v \in R} w(v), \max_{v \in W} w(v), \max_{v \in R} r(v)\right) \qquad \langle w, r, p, j\rangle \hookrightarrow\!\!\!\!\rightarrow^{t} \pi'}{(\sigma, \langle w, r, p, j\rangle) \xhookrightarrow[\mathsf{cycle}\ \mathtt{atom}]{t} \pi'}$$

Fig. 23: Directives of the immediate semantics

reading are stored in w and r. This information is sufficient to check the data-dependencies of an atomic instruction we are trying to fetch w.r.t. the other instructions already in the pipelines. The third component, p, is the delay for each pipeline until it is ready to fetch an instruction, i.e. until its first stage is empty. Since instructions progress by one stage per cycle, this delay is at most one. Finally, j is the delay in cycles until a jump will be retired — remember that at most one jump can be in the pipelines at any time.

Directives The new semantics uses the three same directives as the concrete big-step semantics, fetch `atom`, cycle and cycle `atom`, but adapts them to its new pipeline state. The rules defining these directives are given in Fig. 22 and 23.

Fetch The judgment $(\sigma, \pi) \xhookrightarrow[\mathsf{fetch}\ \mathtt{atom}]{} (\sigma', \pi')$ updates the pipeline state according to rule Fetch. Each component of π is updated by its own rule, and the state σ is immediately updated by applying `atom`.

- the delay of the pipeline X receiving `atom` in its first stage is set to one cycle;
- the delays of all variables written by `atom` are equal to the latency of `atom`;
- the delay of any variable read by `atom` is the maximum between the previous delay and the latency of `atom`.

Notice that the locations updated in w and r are selected with respect to the resolved instruction.

The fetch directive for a jump is defined separately in rule Fetch Jump since it is the only instruction updating the component j. Notice that a jump does not write or read any location after its resolution and w and r remain unchanged.

Cycle In the small-step semantics, executing a cycle consists in making all the instructions progress one stage further in their pipeline. In terms of delays in the immediate semantics, executing a cycle through judgment $\pi \hookrightarrow\!\!\!\!\rightarrow^{1} \pi'$ means that

$$\textsc{Atomic}\quad \frac{(\sigma,\pi) \xhookrightarrow[\text{cycle } \texttt{atom}]{t} \pi'' \qquad (\sigma,\pi'') \xhookrightarrow[\text{fetch } \texttt{atom}]{} (\sigma',\pi')}{(\texttt{atom},\langle\sigma,\pi,h\rangle) \downarrow^{t} \langle\sigma',\pi',h\rangle}$$

$$\textsc{Done}\quad \frac{(s,\langle\sigma,\langle\lambda x.0,\lambda x.0,\lambda X.0,0\rangle,h\rangle) \downarrow^{t} \langle\sigma',\pi',\_\rangle \qquad \pi' \hookrightarrow\!\!\!\!\rightarrow^{t'} \langle\lambda x.0,\lambda x.0,\lambda X.0,0\rangle}{(s,\sigma,h) \downarrow_{t+t'} \sigma' \checkmark}$$

$$\textsc{Cond-True-Correct}\quad \frac{(\texttt{jmp}(b),\omega) \downarrow^{t} \langle\sigma,\pi,h\rangle \qquad \sigma(b)\neq 0 \qquad \neg\,\text{BP-predict}(\ell,h) \qquad h' = \text{BP-update}(\ell,h,false) \qquad (s_1,\langle\sigma,\pi,h'\rangle) \downarrow^{t'} \omega'}{(\ell : \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2,\omega) \downarrow^{t+t'} \omega'}$$

$$\textsc{Cond-True-Incorrect}\quad \frac{(\texttt{jmp} : b,\omega) \downarrow^{t} \langle\sigma,\pi,h\rangle \qquad \sigma(b)\neq 0 \qquad \text{BP-predict}(\ell,h) \qquad h' = \text{BP-update}(\ell,h,false) \qquad \pi \hookrightarrow\!\!\!\!\rightarrow^{|\texttt{jmp}|} \pi' \qquad (s_1,\langle\sigma,\pi',h'\rangle) \downarrow^{t'} \omega'}{(\ell : \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2,\omega) \downarrow^{t+|\texttt{jmp}|+t'} \omega'}$$

Fig. 24: Selected rules of the immediate big-step semantics

all first stages of the pipelines become empty: for any pipeline X, $p(X) = 0$ after a cycle. Similarly, since all the instructions progress one stage closer to the end of their pipeline, the delays before release of all the locations is reduced by one (without going below 0). The judgment is extended to allow for the execution of any number of cycles (including none), and is defined by rule Cycle.

Cycle for atom The judgment $(\sigma,\pi) \xhookrightarrow[\text{cycle } \texttt{atom}]{t} \pi'$ states that π needs to execute t cycles to be able to fetch `atom`, and that after those t cycles the pipeline state will be π'. Once we determined the set of locations needed by `atom`, the delay t is immediately found as the maximum of all the delays on these locations and of the delay until at least one pipeline handling `atom` has its first stage free. In rule Cycle-Release, one can recognize the three types of locks, for instance $\max_{v\in R} w(v)$ corresponds to the RaW lock: `atom` can only be fetched when all locations v read by `atom` are no longer in writing, $w(v) = 0$. Again a specific rule Cycle-Release-Jump is defined for the jump.

Big Step semantics Since we no longer need to represent speculative executions, we drop the continuation style and simply use a big-step semantics. This big-step semantics computes the cost of a whole program by relying on the directives to get the cost of atomic instructions. Similarly to the small-step semantics, we defined both the *fetch cost* and the *execution cost* of a program. Some selected semantics rules are given in Fig. 24, and described below.

Fetch cost The judgment $(s,\omega) \downarrow^{t} \omega'$ states that, starting from the processor state ω, we fetch the program s in t cycles, and end in the processor state ω'.

Atomic Rule Atomic states that the fetch cost of an atomic instruction is the number of cycles necessary for the processor state to be ready for `atom`. This

$$\textsc{Atomic}\ \frac{(\sigma,\pi) \xhookrightarrow[\text{cycle } \texttt{atom}]{t} \pi'' \qquad (\sigma,\pi'') \xhookrightarrow[\text{fetch } \texttt{atom}]{} (\sigma',\pi')}{(\texttt{atom},\langle\sigma,\pi,h\rangle) \downarrow^{t} \langle\sigma',\pi',h\rangle} \qquad \textsc{Skip}\ \frac{}{(\texttt{skip},\omega) \downarrow^{0} \omega} \qquad \textsc{Seq}\ \frac{(s_1,\omega) \downarrow^{t} \omega'' \qquad (s_2,\omega'') \downarrow^{t'} \omega'}{(s_1;s_2,\omega) \downarrow^{t+t'} \omega'}$$

$$\textsc{Cond-True-Correct}\ \frac{\begin{array}{c}(\texttt{jmp}(b),\omega) \downarrow^{t} \langle\sigma,\pi,h\rangle \\ \sigma(b) \neq 0 \qquad \neg\,\text{BP-predict}(\ell,h) \\ h' = \text{BP-update}(\ell,h,false) \\ (s_1,\langle\sigma,\pi,h'\rangle) \downarrow^{t'} \omega'\end{array}}{(\ell : \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2,\omega) \downarrow^{t+t'} \omega'}$$

$$\textsc{Cond-True-Incorrect}\ \frac{\begin{array}{c}(\texttt{jmp} : b,\omega) \downarrow^{t} \langle\sigma,\pi,h\rangle \\ \sigma(b) \neq 0 \qquad \text{BP-predict}(\ell,h) \\ h' = \text{BP-update}(\ell,h,false) \\ \pi \hookrightarrow\!\!\!\!\rightarrow^{|\texttt{jmp}|} \pi' \qquad (s_1,\langle\sigma,\pi',h'\rangle) \downarrow^{t'} \omega'\end{array}}{(\ell : \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2,\omega) \downarrow^{t+|\texttt{jmp}|+t'} \omega'}$$

$$\textsc{While}\ \frac{(\ell : \texttt{if } b \texttt{ then } (s;\ell : \texttt{while } b \texttt{ do } s \texttt{ done}),\omega) \downarrow^{t} \omega'}{(\ell : \texttt{while } b \texttt{ do } s \texttt{ done},\omega) \downarrow^{t} \omega'}$$

$$\textsc{Done}\ \frac{(s,\langle\sigma,\lambda x.0,\lambda x.0,\lambda X.0,0\rangle,h\rangle) \downarrow^{t} \langle\sigma',\pi',\_\rangle \qquad \pi' \hookrightarrow\!\!\!\!\rightarrow^{t'} \langle\lambda x.0,\lambda x.0,\lambda X.0,0\rangle}{(s,\sigma,h) \downarrow_{t+t'} \sigma'\checkmark}$$

Fig. 25: Immediate big-step semantics of a multi-pipelined processor.

amount of cycles is obtained and applied through the cycle `atom` directive. Then, the directive fetch is called to update the delays and the memory state σ.

Conditional We display in Fig. 24 the Cond-True-Correct and Cond-True-Incorrect rules, which handle conditionals when the condition holds. The rules premises are similar to their small-step semantics counter-parts. First, we need t cycles to fetch the `jmp`. This yields a state σ, from which we obtain the value of the register b. In the case of a correct prediction, the branch s_1 starts its execution. Once the prediction is confirmed, $|\texttt{jmp}|$ cycles later, the history is updated. As in the small-step semantics, the history remained unchanged during the speculation. Since the state (σ,π) is not modified once the prediction is confirmed, it is as if we simply fetched s_1 on $\langle\sigma,\pi,h'\rangle$. This yields the premise of rule Cond-True-Correct. In the case of a misprediction, $|\texttt{jmp}|$ cycles are executed, but once the prediction is invalidated all that was fetched and retired is roll backed. From a external point of view, it is as if we simply executed $|\texttt{jmp}|$ cycles, without fetching anything. This is the premise of rule Cond-True-Incorrect, which then start the fetching of the correct branch.

Execution Cost Similarly to the small-step semantics, the execution cost of a program s is the sum of its fetch cost and the cost to empty its pipelines, translated here as the cost to obtain a pipeline state with null delays. The judgment

$$\textsc{Block}\ \frac{s \text{ a block} \qquad [\![\mathsf{blk}]\!]^\sharp \sigma_1^\sharp = (u, o, {\sigma_2}^\sharp) \qquad \sigma_2 \in \mathbb{S}[\![s]\!]\sigma_1}{(s, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, {\sigma_2}^\sharp)}$$

$$\textsc{Seq-No-Block}\ \frac{\begin{array}{c} s_1; s_2 \text{ not a block} \\ (s_1, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, {\sigma_2}^\sharp) \\ (s_2, \sigma_2, {\sigma_2}^\sharp) \Downarrow_{[u',o']} (\sigma_3, {\sigma_3}^\sharp) \end{array}}{(s_1; s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u+u',o+o']} (\sigma_3, {\sigma_3}^\sharp)}$$

$$\textsc{Cond-True}\ \frac{[\![b]\!]\sigma_1 \neq 0 \qquad (\texttt{jmp}(b); s_1, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,\_]} (\sigma_2, {\sigma_2}^\sharp) \qquad (s_1, \sigma_1, \sigma_1^\sharp) \Downarrow_{[\_,o]} (\sigma_2, \sigma_2^\sharp)}{(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o+|\texttt{jmp}|]} (\sigma_2, {\sigma_2}^\sharp)}$$

$$\textsc{Cond-True}\ \frac{[\![b]\!]\sigma_1 = 0 \qquad (\texttt{jmp}(b); s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,\_]} (\sigma_2, {\sigma_2}^\sharp) \qquad (s_1, \sigma_2, \sigma_1^\sharp) \Downarrow_{[\_,o]} (\sigma_2, \sigma_2^\sharp)}{(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o+|\texttt{jmp}|]} (\sigma_2, {\sigma_2}^\sharp)}$$

$$\textsc{While}\ \frac{(\texttt{if } b \texttt{ then } (s; \texttt{while } b \texttt{ do } s \texttt{ done}), \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, {\sigma_2}^\sharp)}{(\texttt{while } b \texttt{ do } s \texttt{ done}, \sigma, \sigma^\sharp) \Downarrow_{[u,o]} (\sigma_2, {\sigma_2}^\sharp)}$$

Fig. 26: The big-step approximate semantics are based on the big-step semantics of our immediate semantics

$(s, \omega) \downarrow_t \sigma' \checkmark$ states that starting from the processor state ω, we execute the program s in t cycles, and end in the memory state σ'. The judgment is obtained from rule Done.

F Approximate semantics

Cost-Approximate Semantics We first show how to lift a cost-approximate semantics on blocks to whole programs. Then we show how to compute blocks cost-approximation.

Cost-Approximate Semantics Bounds on blocks can be computed by abstracting away the memory state σ, but for the whole program it is inadvisable since σ influences the control-flow. For example, the cost of `while(i<len) do blk` is essentially upper-bounded by the maximal cost of blk times the number of loops iterations, which depends on the values of `i` and `len`.

We now build our cost-approximate semantics $(s, \sigma, \sigma^\sharp) \Downarrow_{[u,o]} (\sigma_1, \sigma_1^\sharp)$. Fig. 26 defines the semantics rules for the cost-approximate semantics.ashed changes

Block Code blocks bounds are computed using the cost-approximate block semantics $[\![\mathsf{blk}]\!]^\sharp$. The final memory state is computed with a standard sequential semantics: here we lift the notation $\mathbb{S}[\![\texttt{atom}]\!]\sigma = \sigma'$ to whole blocks.

Sequence For a sequence $s_1; s_2$ of statements which are not atomic (e.g. because s_1 or s_2 contains a while loop), we use the usual rule for sequences defined by Seq-No-Block and sum the costs of both statements. This is valid thanks to Lemma 1.

Conditional The concrete small-step semantics detail the behavior of the processor in case of branch prediction. When concerned about cost, we can simplify the rules: we can compute the result of the predicate b to determine which branch must be taken and compare it with the prediction made with h. In case of misprediction, the program suffers a backtrack penalty, then it executes the correct branch, starting from a pipeline state where all instructions are in stages X_i where $i \geq |\texttt{jmp}|$: the previous instructions have progressed during the execution of the jump. In case of a correct prediction, the jump and the branch are fetched in sequence, without penalty. In that case, the jump is in J_1 and may prevent some instructions to be fetched in J, delaying the execution. We rely on Lemma 1 to ensure that the cost of the two scenario can be bounded. The worst case for the correct prediction is when $|\texttt{jmp}|$ cycles are lost because the branch also start with a conditional (or a loop) and the jumps cannot be executed simultaneously. But this is actually the best case for the misprediction because the $|\texttt{jmp}|$ cycles will always be lost by the backtracking. Thus the over-approximation bound $o + |\texttt{jmp}|$ in Cond-True and Cond-False. The best case of the misprediction is under-bounded by $|\texttt{jmp}|$ plus the minimal cost of the branch, which is always worse than or equal to any case of the correct prediction. The best case for the correct prediction, is the minimal cost of fetching the jump and the branch in sequence, as stated in rules Cond-True and Cond-False.

Theorem 3 states the soundness of our cost-approximate semantics w.r.t. the immediate semantics. It is proved by induction on the program syntax and using some preliminary results on the cost computed from an empty pipeline state.

G Proofs of equivalence

This section proves the equivalence of the three semantics in term of cost and final state σ'. The following theorems, from Theorem 5 to 8, constitute the proof of Theorem 3: the bounds of the cost approximate semantics are sound.

Theorem 5 (Small-step and Concrete semantics equivalence). *Let s be a program, σ a memory state and h a branch history. Then*

$$(s, \sigma, h) \Downarrow_t \sigma' \iff (s, \sigma, h) \overset{CB}{\Downarrow}_t \sigma' \checkmark$$

Theorem 6 (Concrete and Immediate semantics equivalence). *Let s be a program, σ a memory state and h a branch history. Then*

$$(s, \sigma, h) \overset{CB}{\Downarrow}_t \sigma' \checkmark \iff (s, \sigma, h) \downarrow_t \sigma' \checkmark$$

Theorem 7 (Approximate semantics soundness w.r.t. Immediate semantics). *Let s be a program, $\sigma^\sharp$ an alias abstract state, and $\sigma \in \gamma_a(\sigma^\sharp)$ a memory state,s' the instrumentation of s: $(s', \_) = \mathbb{T}(s, \sigma_1^\sharp)$, then*

$$\forall t, h, u, o, \sigma_2, \begin{pmatrix} (s, \sigma_1, h) \downarrow_t \sigma_2 \checkmark \\ \wedge\ (s, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u,o]} (\sigma_2, \_) \end{pmatrix} \implies \begin{pmatrix} \sigma_2[\mathsf{cost} \mapsto t] \in \mathbb{S}[\![s']\!]\sigma_1 \\ \wedge\ u \leq t \leq o \end{pmatrix}$$

Theorem 8 (Approximate semantics existence w.r.t. Immediate semantics). *Let s be a program, $\sigma^\sharp$ an alias abstract state, and $\sigma \in \gamma_a(\sigma^\sharp)$ a memory state,*

$$\forall h, t, (s, \sigma, h) \downarrow_t \sigma_1 \checkmark \implies \exists u, o,\ (s, \sigma, \sigma^\sharp) \Downarrow_{[u,o]} (\sigma_1, \_)$$

Bi-simulation proofs The equivalence proofs should be made by bi-simulation, to guarantee that if a semantics makes a step (apply a directive, execute a cycle, etc) then the other semantics can also make the step, and their final states are in relation. We alleviate such type of proof since the semantics are fully deterministic. Applying any directive will always have one outcome at most. In all our lemmas the initial states are in relation so we simply prove that if any of them can execute a step they both can (section *Step condition* in the proofs). Then we can compare their unique final states to ensure the relation (section *Final states relation* in the proofs).

H Small-step to Concrete semantics

In this section we want to prove that the concrete big-step semantics is equivalent to the small-step semantics in terms of cost, as stated by Theorem 5. First, we define a relation of simulation between the states of the two semantics. Then we prove that the directives to fetch instructions and execute cycles preserves the relation. Finally we prove Theorem 5 by induction on the derivation tree of the execution of the program.

H.1 Relation of simulation

A proof by simulation requires to define a relation of simulation between states of the small-step semantics and the ones of the concrete semantics.

Definition 2 (Small-step - Concrete Relation of simulation). *Let $\omega_S = \langle \sigma_S, \pi_S, h, \beta \rangle$ be a small-step state and $\omega_C = \langle \sigma_C, \pi_C, h \rangle$ a concrete big-step one. ω_S and ω_C are in a simulation relation, noted $\omega_S \overset{SC}{\backsim} \omega_C$, if and only if*

- *Pipeline stages are the same, save for the timestamp:*

$$\forall X_i, \exists k, \pi_S(X_i) = (k, \texttt{atom}) \iff \pi_C(X_i) = \texttt{atom}$$

- *Applying the buffer on σ_S results in σ_C: $\sigma_C = \beta(\sigma_S)$*

H.2 Directives preserves the relation

In both semantics the execution cost is only obtained through a rule Done which decomposed the cost into the fetch cost and the execution cost (to empty the pipelines). We prove that the execution of cycles preserves the relation in the following lemma.

Lemma 2 (Cycle preservation). *Executing cycles preserves the simulation relation.*

$$\begin{array}{ccc} \omega_S & \xrightarrow{\quad} {}^{t} & \omega'_S \\ \vdots\, {\scriptstyle SC} & & \vdots\, {\scriptstyle SC} \\ \omega_C & \xrightarrow{\quad} {}^{t} & \omega'_C \end{array}$$

Corollary 1. *Let ω_S and ω_C be two state in a simulation relation, then both semantics requires the same amount of cycles to empty their pipelines.*

$$\begin{array}{ccc} \omega_S & \xrightarrow{\quad} {}^{t} & \langle \sigma, \pi_\epsilon, h, \emptyset \rangle \\ \vdots\, {\scriptstyle SC} & & \vdots\, {\scriptstyle SC} \\ \omega_C & \xrightarrow{\quad} {}^{t} & \langle \sigma, \pi_\epsilon, h \rangle \end{array}$$

Proof. Each semantics has exactly one rule to execute cycle, without premises that may block the execution. First, let us compare the pipeline states π'_S and π'_C. They are obtain through the application of next and next$'$ on each of their stage. Both functions act similarly and trivially preserve the equivalence. Now let us check the buffer and memory states. By hypothesis, $\sigma_C = \beta(\sigma_S)$. Before considering the buffer β', we prove that $\omega'_C \overset{SC}{\backsim} \langle \sigma_S, \pi_S, h, \beta \cup \text{retired}(\pi_S) \rangle$, that is the states are in relation before the commit. We need to show that

$$\sigma'_C = \bigcirc_{a' \in \text{retired}'(\pi_C)} \mathbb{S}[\![a']\!](\beta(\sigma_S)) = (\beta \cup \text{retired}(\pi_S))(\sigma_S)$$

By relation, the sets retired(π_S) and retired$'(\pi_C)$ are equal if we drop the timestamps. The equality above is thus only a matter of permuting the instruction applications.

All the instructions in retired(π_C) are independents so they can be ordered by decreasing timestamps. Let us suppose that $(i, \texttt{atom}_i) \in \text{retired}(\pi_S)$ and $(j, \texttt{atom}_j)$ is in β with $i < j$. We can permute their application because they cannot have data-dependencies. Indeed $\texttt{atom}_j$ is more recent than $\texttt{atom}_i$ but $\texttt{atom}_i$ just left the pipelines. Thus $\texttt{atom}_j$ was fetched while $\texttt{atom}_i$ was in the pipelines and thus they cannot have data-dependencies. Since their applications can be permuted, it is possible to reorder all the application by decreasing timestamps and thus ensuring the equality.

Committing an instruction trivially preserves the relation, thus executing cycles preserves the relation too.

Lemma 3 (Readiness equivalence). *Let* $\omega_S = \langle \sigma_S, \pi_S, h, \beta \rangle$ *and* $\omega_C = \langle \sigma_C, \pi_C, h \rangle$ *be two states in a simulation relation, then for any instruction* `atom`*, their readiness is equivalent.*

$$\mathrm{ready}(\texttt{atom}, \beta(\sigma_S), \pi_S) \iff \mathrm{ready}(\texttt{atom}, \sigma_C, \pi_C)$$

Proof. The proof is trivial by definition of the rule Ready in both semantics and the relation of simulation stating that $\beta(\sigma_S) = \sigma_C$.

Lemma 4 (Fetch preservation). *Let* $\omega^S = \langle \sigma_S, \pi_S, h, \beta \rangle$ *and* $\omega^C = \langle \sigma_C, \pi_C, h \rangle$ *be two states in a simulation relation,* $\omega^S \overset{SC}{\frown} \omega^C$*, and let* `atom` *be an instruction such that* ω^S *is ready to fetch* `atom` *(and so* ω^C *too by Lemma 3):*

$$\mathrm{ready}(\texttt{atom}, \beta(\sigma^S), \pi^S) \quad \textit{and} \quad \mathrm{ready}(\texttt{atom}, \sigma^C, \pi^C)$$

Let i *be the next timestamp for* ω^S*:* $i = \max(\beta, \pi^S) + 1$*. Then the fetch directive preserves the simulation relation.*

$$\begin{array}{ccc} \omega_S & \xrightarrow[\text{fetch } (i,\ \texttt{atom})]{} {}^{t} & \omega'_S \\ \overset{SC}{\frown} & & \overset{SC}{\frown} \\ \omega_C & \xrightarrow[\text{fetch } \texttt{atom}]{CB} {}^{t} & \omega'_C \end{array}$$

Note that the fetch directive applied on ω^S is actually the fetch directive applied on $(\beta(\sigma^S), \pi^S)$.

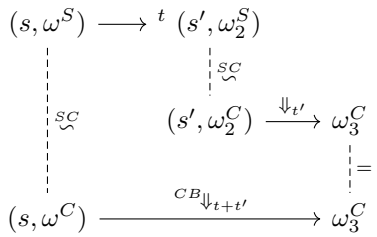
Proof. The proof is trivial by definition of rule Fetch in both semantics and the relation of simulation stating $\beta(\sigma_S) = \sigma_C$.

H.3 Execution cost equivalence

The two semantics do not have the same fetch cost due to the speculation. Indeed, the speculation imposes the execution of cycles even when the continuation has been depleted, which correspond to more cycles than for the fetch cost in the big-step semantics. Thus we only prove the equivalence of the execution cost. Note that in the concrete big-step semantics, the cost of s; `skip` is trivially equal to the cost of s.

Theorem 5 is proved through a lemma stating that whatever small-step the semantics made, the big-step can terminate it and the sum of the costs is the cost of the big-step on the whole statement.

Lemma 5 (Split execution). *Let* ω^S *and* ω^C *be two states in a simulation relation and let* s *be a sequence* $s_1; s_2$*.*



If $s' = \texttt{skip}$ then the cost t' is simply the cost to empty the pipelines, thus $t+t'$ is the execution cost in both semantics. Hence proving this lemma is enough to prove Theorem 5.

Proof. The proof is made by induction on the derivation tree of the statement $(s_1; s_2, \omega^S) \to^t (s', \omega_2^S)$. We suppose that $(s', \omega_2^C)\ {}^{CB}\Downarrow_{t'} \omega_3^C$ ✓ and decompose this cost into the fetch cost t_f and the cost to empty the pipelines t_e, with an intermediate state ω_4^C:

$$(s', \omega_2^C)\ {}^{CB}\Downarrow^{t_f} \omega_4^C \qquad \omega_4^C \hookrightarrow^{t_e} \langle \_, \pi_\epsilon, \_ \rangle \qquad t' = t_f + t_e$$

Atomic If the rule Atomic was applied, then $s_1 = \texttt{atom}$, $t = 0$, $s' = s_2$ and the state ω^S was ready for the instruction `atom`. We note $\omega^S = \langle \sigma^S, \pi^S, h, \beta \rangle$.

$$(\texttt{atom}; s_2, \omega^S) \to^0 (s_2, \omega_2^S) \quad \text{and} \quad \text{ready}(\texttt{atom}, \beta(\sigma^S), \pi^S)$$

By Lemma 3, the big-step state ω^C is ready too. By Lemma 4, the fetch directive preserves the relation. Thus noting $\omega^C = \langle \sigma, \pi, h \rangle$ and $\omega_2^C = \langle \sigma, \pi'', h \rangle$.

$$\textsc{Atomic}\ \frac{\textsc{No-Cycle}\ \dfrac{\text{ready}(\texttt{atom}, \sigma, \pi)}{(\sigma, \pi) \xrightarrow[\text{cycle atom}]{CB\quad 0} (\sigma, \pi)} \qquad (\sigma, \pi) \xrightarrow[\text{fetch atom}]{CB} \pi''}{(\texttt{atom}, \langle \sigma, \pi, h \rangle)\ {}^{CB}\Downarrow^0 \langle \sigma, \pi'', h \rangle}$$

$$\textsc{Seq}\ \frac{\textsc{Atomic}\ \dfrac{\cdots}{(\texttt{atom}, \omega^C)\ {}^{CB}\Downarrow^0 \omega_2^C} \qquad \dfrac{\text{(hypothesis)}}{(s_2, \omega_2^C)\ {}^{CB}\Downarrow^{t_f} \omega_4^C}}{(\texttt{atom}; s_2, \omega^C)\ {}^{CB}\Downarrow^{0+t_f} \omega_4^C}$$

Finally applying the rule Done ensures that $(\texttt{atom}; s_2, \omega^C)\ {}^{CB}\Downarrow_{t'} \omega_3^C$ ✓

Cycle If the rule Cycle was applied, then $s_1 = \texttt{atom}$, $t = 1$ and the state ω^S was not ready for the instruction `atom`. By Lemma 3, ω^C is not ready either.

$$(\texttt{atom}; s_2, \omega^S) \to^1 (\texttt{atom}; s_2, \omega_2^S)$$

By hypothesis, ω_2^C can fetch $\texttt{atom}; s_2$ in t_f cycles. This cost is obtained by first fetching $\texttt{atom}$, we note t'' the cost of this fetch, then by fetching s_2 in $t_f - t''$ cycles. So the following derivation tree can deduce the fetch cost in the concrete semantics.

$$\textsc{Atomic}\ \frac{\textsc{Cycle-Release}\ \dfrac{\begin{array}{c} t''+1>0 \\ (\sigma,\pi) \hookrightarrow^{t''} (\sigma'',\pi'') \\ (\sigma'',\pi'') \hookrightarrow (\sigma',\pi') \\ \neg\,\mathrm{ready}(\texttt{atom},\sigma'',\pi'') \\ \mathrm{ready}(\texttt{atom},\sigma',\pi') \end{array}}{(\sigma,\pi) \xhookrightarrow[\mathsf{cycle}\ \texttt{atom}]{CB\quad t''+1} (\sigma,\pi)} \qquad (\sigma,\pi) \xhookrightarrow[\mathsf{fetch}\ \texttt{atom}]{CB} \pi''}{(\texttt{atom},\langle\sigma,\pi,h\rangle)\ {}^{CB}\Downarrow^{t''+1}\ (\sigma,\omega'')}$$

$$\textsc{Seq}\ \frac{\textsc{Atomic}\ \dfrac{\cdots}{(\texttt{atom},\omega^C)\ {}^{CB}\Downarrow^{t''+1}\ (\sigma,\omega'')} \qquad \dfrac{\text{(hypothesis)}}{(s_2,\omega'')\ {}^{CB}\Downarrow^{t_f-t''}\ \omega_4^C}}{(\texttt{atom}; s_2,\omega^C)\ {}^{CB}\Downarrow^{1+t_f}\ \omega_4^C}$$

Again, the Done rule allows to conclude that the execution in the concrete semantics is $1 + t'$ as expected.

Seq The rule Seq simply performs a re-parenthesizing, and the property can be proved by the induction hypothesis.

Steps If the rule Steps is applied, then there exists t_1, t_2 and s'' and ω_5^S such that $(s_1; s_2, \omega) \to^{t_1} (s'', \omega_5^S)$ and $(s_2, \omega_5^S) \to^{t_2} (s', \omega_2^S)$ and $t = t_1 + t_2$. We can build a $\omega_5^C \overset{SC}{\backsim} \omega_5^S$, by dropping the timestamps and applying the content of the speculation buffer on memory.
By induction hypothesis, since $(s', \omega_2^C)\ {}^{CB}\Downarrow_{t'}\ \omega_5^C$✓, then $(s_2, \omega_5^C)\ {}^{CB}\Downarrow_{t_2+t'}$ ω_3^C✓. Then again by induction hypothesis $(s_1; s_2, \omega^C)\ {}^{CB}\Downarrow_{t_1+t_2+t'}\ \omega_3^C$✓.

Skip This case is trivial, here $\omega^S = \ \omega_2^S$ and thus $\omega^C = \omega_2^C$.

$$\textsc{Seq}\ \frac{\textsc{Skip}\ \dfrac{}{(\texttt{skip},\omega^C)\ {}^{CB}\Downarrow^{0}\ (\sigma,\omega^C)} \qquad \dfrac{\text{(hypothesis)}}{(s,\omega^C)\ {}^{CB}\Downarrow^{t_f}\ \omega_4^C}}{(\texttt{skip}; s_2,\omega^C)\ {}^{CB}\Downarrow^{0+t_f}\ \omega_4^C}$$

Applying the Done rule concludes on the execution cost:

$$(\texttt{skip}; s_2,\omega^C)\ {}^{CB}\Downarrow^{0+t'}\ \omega_3^C$$

Conditional If $s_1 = \texttt{if}\ b\ \texttt{then}\ s_3\ \texttt{else}\ s_4$, then there are two options, either the branch prediction is correct or a misprediction happens. We make the proof only in the case where the condition holds, the other case is symmetrical.

First, a $\texttt{jmp}(b);\texttt{skip}$ is fetched by both semantics. Given that the initial state are in a simulation relation, and following the cases of Atomic and Cycle, one can easily prove that the final state will be in a simulation relation. Because the variable b must be available (not written by an instruction being processed) in both semantics, its valuation should be the same in $\beta(\sigma^S)$ and in σ^C. So both semantics will execute the same branch. Also, the simulation relation imposes the same branch prediction history so the two semantics will make the same prediction.
Let us consider that it is a correct prediction.

$$\textsc{Spec-Cond-True-Correct}$$
$$\frac{\begin{array}{c}(\texttt{jmp}(b);\texttt{skip},\omega^S) \to^{t_1} (\texttt{skip},\langle\sigma_2,\pi_2,h,\beta_2\rangle) \qquad \pi_2(J_1) = (\_,\texttt{jmp}(v)) \\ v \neq 0 \qquad \neg\,\text{BP-predict}(\ell,h) \qquad h' = \text{BP-update}(\ell,h,false) \\ (s_3;s_2,\langle\sigma_2,\pi_2,h,\beta_2\rangle) \xrightarrow{=}{}^{|\texttt{jmp}|} (s',\langle\sigma_3,\pi_3,h,\beta_3\rangle)\end{array}}{(\ell : \texttt{if}\ b\ \texttt{then}\ s_3\ \texttt{else}\ s_4;s_2,\omega^S) \to^{t_1+|\texttt{jmp}|} (s',\langle\sigma_3,\pi_3,h',\beta_3\rangle)}$$

The big-step is able to continue the execution after the $|\texttt{jmp}|$ cycles were executed by the small-step on $s_3;s_2$. The $|\texttt{jmp}|$ cycles were executed even if there were nothing left from $s_3;s_2$. We will distinguish the two cases, corresponding to rules Enforce-Cycles and Enforce-Cycle-Exact.
If rule Enforce-Cycles has been applied we can apply the induction hypothesis on $s_3;s_2$ because no extra cycles were executed. The property can be checked similarly to a sequence, the other premises (branch-prediction and value checked) are equivalent in both semantics.
However, if Enforce-Cycle-Exact was applied, then we decompose the cycles:

$$\textsc{Enforce-Cycle-Exact}$$
$$\frac{(s_3;s_2,\langle\sigma_2,\pi_2,h,\beta_2\rangle) \to^{k} (\texttt{skip},\omega'') \qquad \omega'' \hookrightarrow^{|\texttt{jmp}|-k} \omega'}{(s_3;s_2,\langle\sigma_2,\pi_2,h,\beta_2\rangle) \xrightarrow{=}{}^{|\texttt{jmp}|} (\texttt{skip},\omega')}$$

Let us note $\omega_5^S = \langle\sigma_2,\pi_2,h,\beta_2\rangle$ and let us define $\omega_5^C = \langle\beta_2(\sigma_2),\pi_2',h\rangle$ where π_2' is equal to π_2 but with the timestamps dropped.
From the derivation tree, $s' = \texttt{skip}$ and $t' = 0$. By induction hypothesis, the big-step semantics also fetches the jump in t_1 cycles, and it executes the branch and s_2 in k cycles. Executing cycles preserves the simulation relation

$$\begin{array}{ccc}\omega'' & \hookrightarrow^{|\texttt{jmp}|-k} & \omega' \hookrightarrow^{t'} \omega' \\ \big\vdots\ {\scriptstyle SC}\frown & & \big\vdots\ {\scriptstyle SC}\frown \\ \omega_2^C & \hookrightarrow^{|\texttt{jmp}|-k+t'} & \omega_3^C\end{array}$$

By induction hypothesis applied on $(s_3;s_2,\omega_5^S) \to^k (\texttt{skip},\omega'')$ and the relation above, we have that

$$(s_3;s_2,\omega_5^C)\ {}^{CB}\Downarrow_{k+|\texttt{jmp}|-k+t'}\ \omega_3^C\ \checkmark$$

We separate this execution, and its cost $|\texttt{jmp}|{+}t'$ between the fetch of s_3, the fetch of s_2 and finally emptying the pipelines.

$$\omega_5^C \xrightarrow{s_3 \Downarrow t_3} \omega_6^C \xrightarrow{s_2 \Downarrow t_2} \omega_7^C \hookrightarrow^{t_4} \omega_3^C$$

with $t_3 + t_2 + t_4 = |\texttt{jmp}|{+}t'$.
We trivially have $(\texttt{jmp}(b), \omega^C)\ {}^{CB}\Downarrow^{t_1} \omega_5^C$, thus

$$\textsc{Cond-T.-C.}\ \frac{\dfrac{}{(\texttt{jmp}(b), \omega^C)\ {}^{CB}\Downarrow^{t_1} \langle \sigma, \pi, h\rangle} \quad \dfrac{\text{(hypothesis)}}{(s_3, \langle \sigma, \pi, h'\rangle)\ {}^{CB}\Downarrow^{t_3} \omega_6^C} \quad \pi(J_1) = \texttt{jmp}(v) \quad v \neq 0 \quad \neg\,\text{BP-predict}(\ell, h) \quad h' = \text{BP-update}(\ell, h, false)}{(\ell : \texttt{if}\ b\ \texttt{then}\ s_3\ \texttt{else}\ s_4, \omega)\ {}^{CB}\Downarrow^{t_1+t_3} \omega'}$$

$$\textsc{Seq}\ \frac{\dfrac{\textsc{Cond-True-Correct}\ \ \cdots}{(\ell : \texttt{if}\ b\ \texttt{then}\ s_3\ \texttt{else}\ s_4, \omega)\ {}^{CB}\Downarrow^{t_1+t_3} \omega_6^C} \quad \dfrac{\text{(hypothesis)}}{(s_2, \omega_6^C)\ {}^{CB}\Downarrow^{t_2} \omega_7^C}}{(\ell : \texttt{if}\ b\ \texttt{then}\ s_3\ \texttt{else}\ s_4; s_2, \omega^C)\ {}^{CB}\Downarrow^{t_1+t_3+t_2} \omega_7^C}$$

To empty the pipelines of ω_7^C requires t_4 cycles since ω_3^C is the final state after execution of $s_2; s_3$ and thus its pipelines are empty. The execution cost in the big-step semantics is thus $t_1 + t_3 + t_2 + t_4 = t_1 + |\texttt{jmp}|{+}t'$ which concludes the case of the correct prediction of a jump.
The misprediction is easier and can be proved similarly to a sequence.

While loop The loop is recursively rewritten into a conditional and can be proved similarly, the simulation relation ensures the same branch (loop body or exiting the loop) are taken, hence the same iterations.

This lemma concludes the proof that the small-step can be simulated with the concrete big-step semantics. Since both semantics are deterministic and non-blocking, the concrete big-step semantics can also be simulated by the small-step one, which concludes the proof of Theorem 5.

I Concrete to Immediate semantics

In this section we want to prove that the immediate big-step semantics is equivalent to the concrete big-step semantics in terms of cost, as stated by Theorem 6. The two semantics do not apply the effect of an instruction at the same time. We need to define an equivalence between a concrete state, noted (σ^C, π^C, h^C) and an immediate one (σ^I, π^I, h^I). To define it we consider the application of an instruction currently in π^C on the state σ^C. Like a directive,

we note $(\sigma^C, \pi^C) \xhookrightarrow[\text{apply } X_i]{} (\sigma'^C, \pi^C)$ the application of the instruction currently in X_i:

$$\sigma'^C = \begin{cases} \mathbb{S}[\![\pi^C(X_i)]\!]\sigma^C & \text{if } \pi^C(X_i) \neq \varepsilon \\ \sigma^C & \text{otherwise} \end{cases}$$

Interestingly, the order in which we apply the stages does not change the final state if there is no data-dependencies between the instruction in π^C.

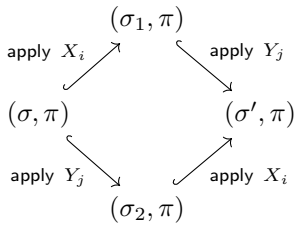
We say that a state is *conflict-free* if it does not contain two instructions that depends on each other in the pipeline state, and we ensure that the directives preserve this property. Then, we ensure that starting with a conflict-free state, the order in which we execute the pipelines is irrelevant. To do so we show that a permutation in the order is irrelevant.

Definition 3. *The rule below defines the cases where π is conflicting. It is said to be* conflict-free *if it is not conflicting.*

$$\textsc{RW Conflict}\ \frac{X_i, Y_j \in \mathsf{Stages} \qquad v \in \mathrm{read} \circ \pi(X_i) \qquad v \in \mathrm{write} \circ \pi(Y_j)}{\pi \text{ is conflicting}}$$

$$\textsc{WW Conflict}\ \frac{X_i, Y_j \in \mathsf{Stages} \qquad v \in \mathrm{write} \circ \pi(X_i) \qquad v \in \mathrm{write} \circ \pi(Y_j)}{\pi \text{ is conflicting}}$$

Lemma 6. *Let (σ, π) be a state such that π is conflict-free. Then for any two stages X_i and Y_j,*



In a conflict-free state, the instruction in X_i and Y_j (if any) do not write in the same locations, and they do not read a location written by the other. Their application can be trivially permuted. Therefore, we can define the equivalence of two states without precising the order of application in the relation $\overset{CI}{\backsim}$ between concrete and immediate states.

Definition 4 (Semantics simulation relation). *A concrete state (σ^C, π) and an instantaneous state $(\sigma^I, \langle w, r, p, j\rangle)$ are in relation if and only if:*

1. *Applying all instructions in π on the state σ^C results in the state σ^I.*

$$(\sigma^C, \pi) \xhookrightarrow[X_1]{} \ldots \xhookrightarrow[Y_n]{} (\sigma^I, \pi) \qquad \textit{with } X_1, \ldots, Y_n = \mathsf{Stages}$$

2. *For any pipeline X, $\pi(X_1) = \varepsilon \iff p(X) = 0$*

3. *For any memory location l, let R be the set of stages X_i that reads at l, $l \in \text{read} \circ \pi(X_i)$, then $r(l) = \max_{X_i \in R}(|\pi(X_i)| - i + 1)$ with the maximum being 0 if $R = \emptyset$.*
4. *For any location x, if there exists (a unique) X_i such that $x \in \text{write} \circ \pi(X_i)$, then $w(x) = |\pi(X_i)| - i + 1$, otherwise $w(x) = 0$.*
5. *For any k, $\pi(J_k) \neq \epsilon \iff j = |\texttt{jmp}| - k + 1$*

This equivalence is noted $(\sigma^C, \pi) \overset{CI}{\backsim} (\sigma^I, \langle w, r, p, j\rangle)$.

In the rest of the section we prove that the directives preserve the relation. We then prove that the big-step semantics of each statement also preserves the relation.

First, we observe an implication of the simulation relation on the variable written.

Lemma 7 (Relation of writings). *Let two states from the two semantics be in relation* $(\sigma^C, \pi) \overset{CI}{\backsim} (\sigma^I, \langle w, r, p, j\rangle)$. *If a location x is not written by any instruction in π then* $\sigma^C(x) = \sigma^I(x)$.

Then, we observe that cycles preserves the simulation relation.

Lemma 8 (Cycle relation). *Let* (σ^C, π) *and* $(\sigma^I, \langle w, r, p, j\rangle)$ *be in relation. Then applying the same number of cycles t will result in states in relation.*

$$\begin{array}{ccc} (\sigma^C, \pi) & \hookrightarrow^t & (\sigma'^C, \pi') \\ \vdots_{CI} & & \vdots_{CI} \\ (\sigma^I, \langle w, r, p, j\rangle) & \twoheadrightarrow^t & (\sigma^I, \langle w', r', p', j'\rangle) \end{array}$$

Proof. **Step condition** There is no condition to apply a step in both semantics.
Final states relation This proof is made my induction on t.

- (Case $t = 0$) In both semantics the final states when executing no cycle are the initial states so the relation is trivially ensured.
- (Case $t > 0$) We suppose that the relation holds after t cycles and we want to prove that it holds after $t + 1$. Remark that in the immediate semantics for any $t \geq 0$.

$$\langle w, r, p, j\rangle \hookrightarrow^t \langle w'', r'', p'', j''\rangle \wedge \langle w'', r'', p'', j''\rangle \hookrightarrow \langle w', r', p', j'\rangle$$

$$\iff$$

$$\langle w, r, p, j\rangle \hookrightarrow^{t+1} \langle w', r', p', j'\rangle$$

So we decompose the application of $t+1$ cycles and we want to prove the relation between the final states on the right in the following diagram. The other relation are ensured by hypothesis and by induction on t. For convenience we include the state σ^I in the rules although executing cycles only involves the pipeline states in the immediate semantics.

$$\begin{array}{ccccc}
(\sigma^C, \pi) & \hookrightarrow & {}^t\,(\sigma''^C, \pi'') & \hookrightarrow & (\sigma'^C, \pi') \\
\downarrow{\scriptstyle CI} & & \downarrow{\scriptstyle CI} & & \downarrow{\scriptstyle CI} \\
(\sigma^I, \langle w, r, p, j\rangle) & \hookrightarrow\!\!\!\!\rightarrow & {}^t\,(\sigma^I, \langle w'', r'', p'', j''\rangle) & \hookrightarrow\!\!\!\!\rightarrow & (\sigma^I, \langle w', r', p', j'\rangle)
\end{array}$$

Let us prove each point of the relation.

1. By induction, (σ''^C, π'') is in relation with $(\sigma^I, \langle w'', r'', p''\rangle)$ so applying all instructions of π'' on σ''^C results in σ^I. The order of these applications can be permuted in our semantics because there should be no data-dependency between the instructions. We thus apply all instructions in $\mathrm{retired}(\pi'')$ first and then the others. Applying all instruction in $\mathrm{retired}(\pi'')$ on σ''^C results exactly in σ'^C. Applying all of the instruction of π'' results in σ^I. If we note S the set of stages which contain an instruction in $\mathrm{retired}(\pi'')$, then

$$(\sigma''^C, \pi'') \xhookrightarrow[\mathsf{apply}\ S]{} (\sigma'^C, \pi'') \xhookrightarrow[\mathsf{apply} \notin S]{} (\sigma^I, \pi'')$$

   As the instructions of π'' not in $\mathrm{retired}(\pi'')$ are exactly the ones in π', we do have that applying all instructions of π' on σ'^C results in σ^I.
2. No matter the states π'' and p'', after executing a cycle $\pi'(X_1) = \epsilon$ by definition of next and $p'(X) = 0$ by rule Cycle so the equivalence trivially holds.
3. Let l be a memory location and R be the set of stages X_i such that $l \in \mathrm{read} \circ \pi''(X_i)$. By induction hypothesis, $r''(l) = \max_{X_i \in R} |\pi''(X_i)| - i + 1$. Again we separate the set R between the stages that will be retired and the others. We note S the first group, such that $X_i \in S \iff |\pi''(X_i)| = i$, and S' the latter. In π', by definition of next, a stage X_i can read l if and only if $\pi'(X_{i-1})$ was reading it and if $|\pi''(X_{i-1})| \neq i - 1$. So X_i reads l in π' iff X_{i-1} is in S'. Let R' be this set of stages, we have $X_{i-1} \in S' \iff X_i \in R'$.

$$\begin{aligned}
r'(l) &= \max(0, r''(l) - 1) \\
&= \max(0, \max_{X_i \in R} |\pi''(X_i)| - i) \\
&= \max(0, \max_{X_i \in S'} |\pi''(X_i)| - i) && \text{by definition of } S \\
&= \max(0, \max_{X_{i-1} \in S'} |\pi''(X_{i-1})| - i - 1) \\
&= \max(0, \max_{X_{i-1} \in S'} |\pi'(X_i)| - i + 1) \\
&= \max(0, \max_{X_i \in R'} |\pi'(X_i)| - i + 1)
\end{aligned}$$

   which concludes the third point of the relation.

4. The property on w' is proved similarly to r' as it is only a particular case where the set of stages is either empty or a singleton.
5. If there is a jump in a stage J_k in π'' then by relation $j'' = |\texttt{jmp}| - k + 1$. If $k = |\texttt{jmp}|$ then $j'' = 1$ and after a cycle the jump leaves the pipeline. In that case, $j' = 0$ and there is no jump in the pipeline as expected. Otherwise, if $k < |\texttt{jmp}|$ then $j' = j'' - 1$. After the cycle, the jump is in pipeline J_{k+1}, thus preserving the relation. Finally, we can consider the case where there is no jump in π'' and where, by relation, $j'' = 0$. The execution of a cycle does not add a jump in π' and $j' = j'' = 0$ thus the relation is preserved.

The final states are in relations and by induction, executing any amount of cycles preserves the relation.

There is no notion of readiness in the immediate semantics since we compute the exact number of cycles to release all the resources needed for `atom`. In the following lemma the readiness of the immediate state is defined as the number of cycles needed being equal to zero. The following lemma also ensures a key aspect of our relation: if both states are ready then the set of locations computed by the read and write functions should be the same, despite the two different states of the memory σ^C and σ^I.

Lemma 9 (Ready in immediate). *Let* `atom` *be an atomic instruction and let* (σ^C, π) *and* $(\sigma^I, \langle w, r, p, j\rangle)$ *be in relation, then*

$$\text{ready}(\texttt{atom}, \sigma^C, \pi) \iff \max\left(\min_{X \in \texttt{atom}} p(X), \max_{v \in R} w(v), \max_{v \in W} w(v), \max_{v \in R} r(v)\right) = 0$$

and

$$\text{ready}(\texttt{atom}, \sigma^C, \pi) \implies \text{write}(\texttt{atom}, \sigma^C) = W \ \wedge \ \text{read}(\texttt{atom}, \sigma^C) = R$$

with $W = \text{write}(\texttt{atom}, \sigma^I)$ *and* $R = \text{read}(\texttt{atom}, \sigma^I)$.

Proof. Let us prove each side of the equivalence.

$\Rightarrow$ We suppose that $\text{ready}(\texttt{atom}, \sigma^C, \pi)$ holds. First let us check the equalities $\text{read}(\texttt{atom}, \sigma^I) = \text{read}(\texttt{atom}, \sigma^C)$ and $\text{write}(\texttt{atom}, \sigma^I) = \text{write}(\texttt{atom}, \sigma^C)$. By point (1) of the relation, applying all instructions of π on σ^C results in σ^I. Since $\text{ready}(\texttt{atom}, \sigma^C, \pi)$, none of these instructions can write a location needed by `atom` (it would violate WaR or WaW dependency). So for any $x \in \text{read}(\texttt{atom}, \pi)$, applying a stage X_i of π on σ^C will not change the value associated to x.

$$\text{ready}(\texttt{atom}, \sigma^C, \pi) \implies \forall x \in \text{read}(\texttt{atom}, \pi), \sigma^C(x) = \sigma^I(x)$$

In the sets computed by read and write, the values in σ are only used to get the value of registers read by `atom`. So

$$\text{read}(\texttt{atom}, \sigma^I) = \text{read}(\texttt{atom}, \sigma^C) \text{ and } \text{write}(\texttt{atom}, \sigma^I) = \text{write}(\texttt{atom}, \sigma^C)$$

Now we need to prove that the delay in the immediate semantics is null. We prove that each component of the maximum is null.

- For the minimum on p, we now that since $\mathrm{ready}(\mathtt{atom}, \sigma^C, \pi)$ holds, there exists a pipeline $X \in \mathtt{atom}$ such that $\pi(X_1) = \varepsilon$. By point (2) of the relation, this implies that $p(X) = 0$ so $\min_{X \in \mathtt{atom}} p(X) = 0$.
- For the maximum of w on $\mathrm{read}(\mathtt{atom}, \sigma^I)$, let us take any $x \in \mathrm{read}(\mathtt{atom}, \sigma^I) = \mathrm{read}(\mathtt{atom}, \sigma^C)$. Then the set of pipelines that write x in π is empty, otherwise the $\mathrm{locks}(\mathtt{atom}, \mathtt{atom}', \sigma^C)$ statement would hold for some $\mathtt{atom}'$ writing x, and the state would not be ready for $\mathtt{atom}$. This set of pipeline being empty, the maximum is defined as 0.
- The case of w on $\mathrm{write}(\mathtt{atom}, \sigma^I)$ and of r on $\mathrm{write}(\mathtt{atom}, \sigma^I)$ are handled similarly, the set of pipelines are empty and thus the maximum is 0.

As each component is null, the maximum is also null which concludes this implication.

$\Leftarrow$ We suppose now that the delay computed by the immediate semantics is null and we want to prove that $\mathrm{ready}(\mathtt{atom}, \sigma^C, \pi)$ holds. We need to ensure the existence of a pipeline for $\mathtt{atom}$ in π and that $\mathrm{locks}(\mathtt{atom}, \mathtt{atom}', \sigma^C)$ cannot hold for any $\mathtt{atom}'$ in π.

First, let us find a pipeline for $\mathtt{atom}$ in π. None of the component of the maximum can be negative so they are all null, in particular, $\min_{X \in \mathtt{atom}} p(X) = 0$ and there exists $X \in \mathtt{atom}$ such that $p(X) = 0$. By point (2) of the relation $\pi(X_1) = \varepsilon$, and so there exists a pipeline in π that can fetch $\mathtt{atom}$.

Then let us ensure that $\mathrm{locks}(\mathtt{atom}, \mathtt{atom}', \sigma^C)$ does not hold for any $\mathtt{atom}'$ in π. We reason by contradiction, let us suppose that there is a RaW or WaW conflict: then there exists a location $x \in \mathrm{read}(\mathtt{atom}, \sigma^C)$ or in $\mathrm{write}(\mathtt{atom}, \sigma^C)$ such that there is a stage X_i satisfying $x \in \mathrm{write} \circ \pi(X_i)$. Then, by point (4) of the relation, $w(x) = |\pi(X_i)| - i + 1 = 0$. So $|\pi(X_i)| = i - 1$, which is impossible in our semantics because the instruction in $\pi(X_i)$ should have been retired. Let us suppose that there is a WaR conflict: then there exists a location $x \in \mathrm{write}(\mathtt{atom}, \sigma^C)$ such that there is a stage X_i satisfying $x \in \mathrm{read} \circ \pi(X_i)$. By point (3) of the relation, $r(x) = 0 \geq |\pi(X_i)| - i + 1$ and similarly to the previous case it is impossible because the instruction should have been retired. So there is no locks possible and $\mathrm{ready}(\mathtt{atom}, \sigma^C, \pi)$ holds.

A similar statement can be made in case of a `jmp`, where the j component is used. We admit that the lemma above cover this case.

Lemma 10 (Cycle for atomic relation). *Let* $\mathtt{atom}$ *be an atomic instruction and let* (σ^C, π) *and* $(\sigma^I, \langle w, r, p, j \rangle)$ *be in relation. Then the number of cycles needed by both states to fetch* $\mathtt{atom}$ *is the same and the final states are in relation.*

$$\begin{array}{ccc} (\sigma^C, \pi) & \xhookrightarrow[\text{cycle } \mathtt{atom}]{t} & (\sigma'^C, \pi') \\ \vdots\, \frown^I & & \vdots\, \frown^I \\ (\sigma^I, \langle w, r, p, j \rangle) & \xhookrightarrow[\text{cycle } \mathtt{atom}]{t} & (\sigma^I, \langle w', r', p', j' \rangle) \end{array}$$

Proof. Lemma 8 ensures that if both semantics execute the same amount of cycles then they remain in relation, so proving the *Final states relation* is trivial. Thus the main goal of this proof is to justify that they both execute t cycles (proof of the *Step condition*). Let us suppose that they do not, that the concrete semantics executes t^C cycles to fetch `atom` and the immediate semantics executes t^I cycles.

Case ($t^C < t^I$) Let us execute t^C cycles on $\langle w, r, p\rangle$. By lemma 8, the states are in relation: $({\sigma'}^C, \pi') \overset{CI}{\backsim} (\sigma^I, \langle w'', r'', p''\rangle)$. After t^C cycles the concrete state is ready so by lemma 9, $\mathrm{read}(\texttt{atom}, \sigma^I) = \mathrm{read}(\texttt{atom}, {\sigma'}^C)$ and $\mathrm{write}(\texttt{atom}, \sigma^I) = \mathrm{write}(\texttt{atom}, {\sigma'}^C)$. We named these two sets R and W as in the lemma.

Let us note $\delta = t^I - t^C > 0$, the number of cycles that the immediate state $\langle w'', r'', p''\rangle$ still needs to execute to be ready to fetch `atom`. Then

$$\delta = \max\left(\min_{X\in\texttt{atom}} p''(X), \max_{v\in R} w''(v), \max_{v\in W} w''(v), \max_{v\in R} r''(v)\right) > 0$$

But by lemma 9, since $(\sigma^C, \pi) \overset{CI}{\backsim} (\sigma^I, \langle w'', r'', p''\rangle$, $\delta = 0$ which is absurd. Thus $t^I \geq t^C$.

Case $t^C > t^I$ Let us execute t^I cycles on (σ^C, π) to get $({\sigma''}^C, \pi'')$, which is in relation with $(\sigma^I, \langle w', r', p'\rangle)$ by lemma 8. The concrete semantics needs more cycles, which implies that $\neg\,\mathrm{ready}(\texttt{atom}, {\sigma''}^C, \pi'')$.

Since $t^I = \max(\min_{X\in\texttt{atom}} p(X), \max_{v\in R} w(v), \max_{v\in W} w(v), \max_{v\in R} r(v))$ then all the components of the maximum are null in $\langle w', r', p'\rangle$, after we executed t^I cycles. It is obvious for the components on w and r since t^I is greater than their maximum. But the case of component $\min_{X\in\texttt{atom}} p'(X) = 0$ is particular. If $t^I \geq 1$ then the nullity is trivial: $p'(X) = 0$ for all X. Otherwise, if $t^I = 0$ then it implies that $\min_{X\in\texttt{atom}} p(X) = 0$, and that $p = p'$. So $\min_{X\in\texttt{atom}} p'(X) = 0$.

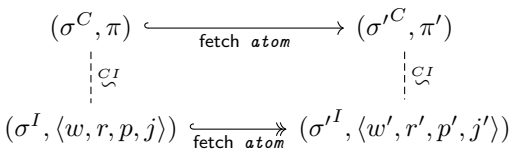
$$\begin{aligned}
&\max\left(\min_{X\in\texttt{atom}} p'(X), \max_{v\in R} w'(v), \max_{v\in W} w'(v), \max_{v\in R} r'(v)\right)\\
=&\max\left(0, \max_{v\in R}\max(w(v) - t^I, 0), \max_{v\in W}\max(w(v) - t^I, 0), \max_{v\in R}\max(r(v) - t^I, 0)\right)\\
=&0
\end{aligned}$$

By lemma 9, π'' must be ready for `atom` too, which is absurd, so $t^C \leq t^I$.

These two absurd cases proves that $t^C = t^I$ which concludes that both semantics can apply the same amount of cycles for `atom`.

The lemma can also be applied to the `jmp` instruction, using the j component.

Lemma 11 (Fetch relation). *Let* `atom` *be an atomic instruction and let* (σ^C, π) *and* $(\sigma^I, \langle w, r, p, j\rangle)$ *be in relation. Let us suppose that* $\mathrm{ready}(\texttt{atom}, \texttt{atom}', \sigma^C)$ *holds, then their final states after fetching it are in relation.*



Proof. **Step condition** Both semantics requires the existence of a pipeline X ready to fetch `atom`. By point (2) of the relation, $p(X) = 0$ if and only if $\pi(X_1) = \varepsilon$ so they will both be able to apply the fetch directive.

Final states relation In the immediate semantics, each component w, r, p and σ^I is updated separately and we can check each point of the relation.

1. The new state of the memory in the immediate semantics is $\sigma'^I = \mathbb{S}[\![\texttt{atom}]\!]\sigma^I$. As for the concrete semantics, σ^C remains the same but we added exactly one instruction in π: resolve$(\texttt{atom}, \sigma^C)$. It is placed in a stage X_1 which was empty, so no instruction is removed from π in π'. By point (1) of the relation on the initial states, applying all instructions of π on σ^C results in σ^I so applying all instructions of π' on σ^C results in $\mathbb{S}[\![\text{resolve}(\texttt{atom}, \sigma^C)]\!]\sigma^I$. So we need to ensure that $\mathbb{S}[\![\text{resolve}(\texttt{atom}, \sigma^C)]\!]\sigma^I = \mathbb{S}[\![\texttt{atom}]\!]\sigma^I$. As the resolve operation replaces the registers read by their value in σ^C, we must ensure that $\sigma^C(x) = \sigma^I(x)$ for any register x read by `atom`. These registers have the same value by point (1) if and only if none of the instruction in π modifies them. Let us suppose that there exists a register read by `atom` and a stage Y_i such that $x \in \text{write} \circ \pi^C(\texttt{atom})$. Then locks$(\texttt{atom}, \pi^C(\texttt{atom}), \sigma^C)$ holds (rule Lock RaW) and ready$(\texttt{atom}, \pi^C(\texttt{atom}), \sigma^C)$ cannot holds which is against our hypothesis. So

$$\forall x \in \text{read}(\texttt{atom}, \sigma^C) \cap \mathsf{Reg}, \sigma^C(x) = \sigma^I(x)$$

and thus

$$\mathbb{S}[\![\text{resolve}(\texttt{atom}, \sigma^C)]\!]\sigma^I = \mathbb{S}[\![\texttt{atom}]\!]\sigma^I$$

Point (1) of the relation holds for (σ^C, π') and σ^I.

2. Both semantics will chose the same pipeline X to fetch `atom` if their states are in relation. Indeed by point (2) of the relation, the set of pipelines $\{Y \in \texttt{atom} \mid \pi(Y_1) = \varepsilon\}$ is the same as $\{Y \in \texttt{atom} \mid p(Y) = 0\}$. The minimum X is the same in both semantics. For any pipeline Y other than X nothing has changed so $\pi(Y_1) = \pi'(Y_1) = \varepsilon \iff p(Y) = p'(Y) = 0$. As for X, we now have $\pi'(X_1) = \text{resolve}(\texttt{atom}, \sigma^C, \pi) \neq \varepsilon$ and $p'(X) = 1$. Point (2) of the relation is thus satisfied by the final states π' and p', for any pipeline.
3. For any location not read by `atom`, nothing as changed in r' and in π' and the property is still valid. Now let us consider $l \in \text{read} \circ \pi'(X_1)$ (where X_1 is the stage where `atom` was put). Let R be the set of stages that

read l in π. Then the set that read l in π' is $R \cup \{X_1\}$.

$$\begin{aligned} r'(l) &= \max(r(l), |\texttt{atom}|) \\ &= \max(\max_{Y_i \in R}(|\pi(Y_i)| - i + 1), |\pi(X_1)|) \\ &= \max(\max_{Y_i \in R}(|\pi(Y_i)| - i + 1), |\pi(X_1)| - 1 + 1) \\ &= \max_{Y_i \in R \cup \{X_1\}}(|\pi(Y_i)| - i + 1) \end{aligned}$$

Thus point (3) is satisfied.

4. The same reasoning applies to point (4).
5. For $\texttt{atom} \neq \texttt{jmp}$, the j component remain unchanged and thus the relation is preserved since no jump is added to the pipelines either. If $\texttt{atom} = \texttt{jmp}(v)$ then $j = 0$ and after the fetch $j' = |\texttt{jmp}| + 1$ while J_1 contains a jump, the relation is also preserved.

Thus for all directives the two semantics preserves the relation. This can be extended to show that the big-step semantics preserves the relation for any statement, by induction on the syntax of the statement. Below we consider the case of the conditional since we must ensure that the two semantics will make the same predictions and that they take the same branch.

Lemma 12 (Atomic preservation). *Any atomic instruction* `atom` *preserves the relation.*

$$\begin{array}{ccc} (\sigma^C, \pi, h) & \xhookrightarrow{(\texttt{atom},\cdot)\,{}^{CB}\Downarrow^t_\cdot} & (\sigma'^C, \pi', h') \\ \vdots\, {}^{CI}_{\frown} & & \vdots\, {}^{CI}_{\frown} \\ (\sigma^I, \langle w, r, p\rangle, h) & \xhookrightarrow{(\texttt{atom},\cdot)\downarrow^t_\cdot} & (\sigma'^I, \langle w', r', p'\rangle, h') \end{array}$$

Proof. Lemmas 10 and 11 are enough to prove this lemma.

Lemma 13 (Conditional preservation). *Let us take b an operand, s_1 and s_2 two statements, then the two big-step semantics applied on the conditional* `if` b `then` s_1 `else` s_2 *preserves the relation.*

$$\begin{array}{ccc} (\sigma^C, \pi, h) & \xhookrightarrow{(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2,\cdot)\,{}^{CB}\Downarrow^t_\cdot} & (\sigma'^C, \pi', h') \\ \vdots\, {}^{CI}_{\frown} & & \vdots\, {}^{CI}_{\frown} \\ (\sigma^I, \langle w, r, p, j\rangle, h) & \xhookrightarrow{(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2,\cdot)\downarrow^t_\cdot} & (\sigma'^I, \langle w', r', p', j'\rangle, h') \end{array}$$

Proof. Let us show that if the concrete semantics makes a step, then so does the immediate semantics. First, let us recall the concrete and immediate semantics rules in case of correct prediction, with some renaming to ease the proof. Each

processor state is indexed. The memory states σ are commons to the two semantics so we add an C or I exponent on these elements, for the concrete and immediate semantics respectively. For the branch predictor history, we keep the same notation without indexes because they are equal in both semantics, going through the same modifications by BP-update.

Cond-True-Correct-Concrete

$$\frac{\begin{array}{c}(\texttt{jmp}(b), \langle \sigma_0^C, \pi_0, h_0\rangle) \ {}^{CB}\Downarrow^{t^C} \ \langle \sigma_1^C, \pi_1, h_1\rangle \qquad \sigma_1^C(b) \neq 0 \\ \neg\,\text{BP-predict}(\ell, h_1) \qquad h_2 = \text{BP-update}(\ell, h_1, false) \\ (s_1, \langle \sigma_1^C, \pi_1, h_2\rangle) \ {}^{CB}\Downarrow^{t'^C} \ \langle \sigma_2^C, \pi_2, h_3\rangle\end{array}}{(\ell : \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \langle \sigma_0^C, \pi_0, h_0\rangle) \ {}^{CB}\Downarrow^{t^C + t'^C} \ \langle \sigma_2^C, \pi_2, h_3\rangle}$$

Cond-True-Correct-Immediate

$$\frac{\begin{array}{c}(\texttt{jmp}(b), \langle \sigma_0^I, \langle w_0, r_0, p_0, j_0\rangle, h_0\rangle) \downarrow^{t^I} \langle \sigma_1^I, \langle w_1, r_1, p_1, |\texttt{jmp}|\rangle, h_1\rangle \qquad \sigma_1^I(b) \neq 0 \\ \neg\,\text{BP-predict}(\ell, h_1) \qquad h_2 = \text{BP-update}(\ell, h_1, false) \\ (s_1, \langle \sigma_1^I, \langle w_1, r_1, p_1, |\texttt{jmp}|\rangle, h_2\rangle) \downarrow^{t'^I} \langle \sigma_2^I, \langle w_2, r_2, p_2, j_2\rangle, h_3\rangle\end{array}}{(\ell : \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \langle \sigma_0^I, \langle w_0, r_0, p_0, j_0\rangle, h_0\rangle) \downarrow^{t^I + t'^I} \langle \sigma_2^I, \langle w_2, r_2, p_2, j_2\rangle, h_3\rangle}$$

Since the concrete semantics can make a step, all the premises of rule Cond-True-Correct-Concrete holds. By hypothesis the initial states are in relation: $\langle \sigma_0^C, \pi_0, h_0\rangle \overset{CI}{\backsim} \langle \sigma_0^I, \langle w_0, r_0, p_0, j_0\rangle, h_0\rangle$. Finally, by induction, we suppose that if the initial processors states are in relations, then they remain in relation and have the same fetch time for sub-program s_1. Let us show that all the premises of rule Cond-True-Correct-Immediate holds and that $t^C = t^I \wedge t'^C = t'^I$.

Jump By lemma 12, and since $\langle \sigma_0^C, \pi_0, h_0\rangle \overset{CI}{\backsim} \langle \sigma_0^I, \langle w_0, r_0, p_0, j_0\rangle, h_0\rangle$ then the two semantics will have the same fetch time for the jump instruction and the final states will be in relation:

$$t^C = t^I \quad \wedge \quad \langle \sigma_1^C, \pi_1, h_1\rangle \overset{CI}{\backsim} \langle \sigma_1^I, \langle w_1, r_1, p_1, |\texttt{jmp}|\rangle, h_1\rangle$$

Conditional value The concrete semantics checks that the value of operand b now that the register has been replaced by value v. The concrete semantics of an atomic fetch imposes that $v = \sigma_1^C(b)$. If b is a constant then $\sigma_1^I(b) = v \neq 0$. Otherwise, i.e. if b is a register, then no instruction in π_1 can be writing in b has it would block the fetch of the jump instruction. So, by lemma 7, $\sigma_1^I(b) = \sigma_1^C(b) \neq 0$.

Branch prediction The two semantics have the same treatment of the branch prediction history.

Branch fetch time The initials states for the branch execution are in relation, thus by induction on the syntax, the two final states are in relation and the two semantics compute the same cost $t' = t'^I = t'^C$

The final state are in relation and the costs computed are equal $t^C + t'^C = t^I + t'^I$, proving the equivalence of the semantics on conditionals.

Lemma 14 (Loop preservation). *Let us take b an operand, s a statement, then the two big-step semantics applied on the loop* `while b do s done` *preserves the relation.*

$$\begin{array}{ccc} (\sigma^C, \pi, h) & \xhookrightarrow{(\texttt{while } b \texttt{ do } s \texttt{ done},\cdot)\,{}^{CB}\Downarrow^t\cdot} & (\sigma'^C, \pi', h') \\ \vdots\, {}^{CI}\wr & & \vdots\, {}^{CI}\wr \\ (\sigma^I, \langle w, r, p, j\rangle, h) & \xhookrightarrow{(\texttt{while } b \texttt{ do } s \texttt{ done},\cdot)\downarrow^t\cdot} & (\sigma'^I, \langle w', r', p', j'\rangle, h') \end{array}$$

Proof. The while loop will be unrolled in as many conditional as necessary to end the iteration. By recurrence on the number of unrolling the lemma can be proved using the previous one on conditionals.

J Immediate semantics to cost-approximate semantics

In this section we want to prove that the cost-approximate semantics is sound w.r.t. the Immediate big-step semantics in terms of cost, as stated by Theorem 7 as well as proving Theorem 8. Theorem 8 is trivially proven by induction on the syntax of the program and we do not detail it.

In a first time, we suppose that the approximation of blocks $[\![\mathsf{blk}]\!]^\sharp$ is sound, but we are more precise than Theorem 2, by differentiating the fetch cost f and the execution cost e.

Lemma 15 (Block approximation soundness). *For any block blk, any abstract alias memory state $\sigma^\sharp$, any bounds u, o:*

$$[\![\mathsf{blk}]\!]^\sharp\sigma^\sharp = (u, o, \_) \;\Rightarrow\; \forall \sigma \in \gamma_a(\sigma^\sharp),\, h,\; (\forall e, (\mathsf{blk}, \langle \sigma, \pi_\epsilon, h\rangle) \downarrow_e \_ \checkmark \Rightarrow e \leq o) \tag{1}$$

$$\wedge\; \big(\forall f, (\mathsf{blk}, \langle \sigma, \pi_\epsilon, h\rangle) \downarrow^f \_ \Rightarrow f \geq u\big) \tag{2}$$

We use this hypothesis to prove that the over-approximation is sound w.r.t. the immediate semantics execution cost, while the under-approximation is sound w.r.t. the fetch cost. The expression of the execution cost t' can be simplified in the immediate semantics. Indeed, given the fetch cost t, and the final pipeline state π'' after the program has been fetched, the execution cost is $t' = t + \max(\pi'')$ since $\max(\pi'')$ is exactly the cost of emptying the pipelines (if all instructions have at least one dependency, which is the case in our language[6]). The following lemma is thus a sub-part of Theorem 7.

Lemma 16. *Let s be a program, $\sigma^\sharp$ an alias abstract memory state and $\sigma \in \gamma_a(\sigma^\sharp)$:*

$$\forall h, t, u, o, \pi', \sigma' \quad \begin{pmatrix} (s, \langle \sigma, \pi_\epsilon, h\rangle) \downarrow^t \langle \sigma', \pi', \_\rangle \\ \wedge\; (s, \sigma, \sigma^\sharp) \Downarrow_{[u,o]} (\sigma', \_) \end{pmatrix} \Rightarrow\; u \leq t \;\wedge\; t + \max(\pi') \leq o$$

[6] In a language where this is not the case we could add artificial dependencies, with temporary registers.

Proof. The proof is inductive on the syntax of s.

Block First, let us consider that s is a block of atomic instructions $a_1; \ldots; a_n$. In that case, the two approximation semantics both rely on the approximation of blocks which is sound by hypothesis. So the proof is trivial.

Sequence Then, let us consider that $s = s_1; s_2$ is a sequence (but not a block). In the immediate semantics, the fetch cost of s is the sum of the fetch costs of s_1 and s_2. The execution cost of s is obtained by adding the cost of emptying the final pipeline, that is $\max(\pi')$.

$$(s_1, \langle \sigma, \pi_\epsilon, h \rangle) \downarrow^t \langle \sigma_1, \pi_1, h_1 \rangle \qquad (s_2, \langle \sigma_1, \pi_1, h_1 \rangle) \downarrow^{t'} \langle \sigma', \pi', \_ \rangle$$

$$(s, \sigma, \pi_\epsilon) \downarrow_{t+t'+\max(\pi')} \sigma' \checkmark$$

In the two approximate semantics, we will also have bounds for s_1 and s_2. We apply the induction hypothesis on s_1 which starts with empty pipelines, the bounds of s_1 are sound.

$$(s_1, \sigma, \sigma^\sharp) \Downarrow_{[u,o]} (\sigma_1, \sigma_1^\sharp)$$

$$u \leq t \quad \text{and} \quad t + \max(\pi_1) \leq o$$

As for s_2, our immediate execution is not from an empty pipeline. The Lemma 1 gives us however additional information: if we note e the cost to fetch s_2 from the empty pipeline, $(s_2, \langle \sigma_1, \pi_\epsilon, h_1 \rangle) \downarrow^e \langle \sigma', \pi_e, \_ \rangle$, then

$$e \leq t' \text{ and } t' + \max(\pi') \leq \max(\pi_1) + e + \max(\pi_e)$$

(It is more efficient to start the execution of s_2 from the current pipeline state π_1 than to empty it first and then to execute s_2 on an empty pipeline.) Thus we have by induction hypothesis that the approximated cost of s_2, noted o' and u' bounds the retire cost of s_2 from the empty pipeline:

$$(s_2, \sigma_1, \sigma_1^\sharp) \Downarrow_{[u',o']} (\sigma', \_)$$

$$u' \leq e \qquad e + \max(\pi_e) \leq o'$$

With this defined, our goal is to prove that

$$u + u' \leq t + t' + \max(\pi') \leq o + o'$$

$$\begin{aligned} u + u' &\leq t + e \leq t + t' \\ &\leq t + t' + \max(\pi') && \triangleleft \text{ Execution cost of the sequence} \\ &\leq t + \max(\pi_1) + e + \max(\pi_e) \\ &\leq o + o' \end{aligned}$$

Conditional Finally, let us treat conditionals as loops are directly derived from them. Let $s = \ell : \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2$. First let us suppose that the branch prediction is correct, and that $\sigma(b) \neq 0$ (the then branch must be taken), the other branch being symmetrical. In that case, the cost of the conditional is $t + t'$ where

$$(\texttt{jmp}(b), \langle \sigma_\epsilon, \pi, h\rangle) \downarrow^t \langle \sigma_1, \pi_1, h\rangle \quad \text{and} \quad (s_1, \langle \sigma_1, \pi_1, h_1\rangle) \downarrow^{t'} \langle \sigma', \pi', h'\rangle$$

and $h_1 = \text{BP-update}(h, \ell, false)$. The proof is thus exactly the same as in the case of the sequence, the value of the branch predictor history being irrelevant. The under-approximated cost of the conditional corresponds exactly to the under-approximated cost of the sequence, while the over-approximation adds a constant $|\texttt{jmp}|$ to the over-approximated cost of the sequence, it is thus still a sound over-approximation.
For the other case, a misprediction, we need to take the backtrack penalty into account. In the immediate semantics, the fetch cost is $t+t'+|\texttt{jmp}|$ given the costs of the immediate semantics

$$(\texttt{jmp}(b), \langle \sigma_\epsilon, \pi, h\rangle) \downarrow^t \langle \sigma_1, \pi_1, h_1\rangle \quad \text{and} \quad (s_1, \langle \sigma_1, \pi_2, h_1\rangle) \downarrow^{t'} \langle \sigma', \pi', h'\rangle$$

$$\pi_1 \hookrightarrow^{|\texttt{jmp}|} \pi_2$$

Interestingly, the max of π_2 is the max of π_1 minus the latency of the jump, which is strictly positive since π_1 has at least the delay to retire the jump it just fetched.

$$\max(\pi_2) = \max(\pi_1) - |\texttt{jmp}|$$

Similarly to the sequence we define the fetch cost e of the branch s_1 from an empty pipeline state.

$$(s_1, \langle \sigma_1, \pi_\epsilon, h_1\rangle) \downarrow^e \langle \sigma', \pi_e, h'\rangle \quad \text{and}$$

with $e \leq t'$ and $t' + \max(\pi') \leq \max(\pi_2) + e + \max(\pi_e)$. The approximated costs are

$$(\texttt{jmp}(b), \sigma, \sigma^\sharp) \Downarrow_{[u,o]} (\sigma_1, \sigma_1^\sharp) \quad \text{and} \quad (s_1, \sigma, \sigma^\sharp) \Downarrow_{[u',o']} (\sigma_1, \sigma_1^\sharp)$$

such that

$$u \leq t \quad \text{and} \quad t + \max(\pi_1) \leq o$$
$$u' \leq e \quad \text{and} \quad e + \max(\pi_e) \leq o'$$

Then, similarly to the sequence

$$\begin{aligned}
u + u' &\leq t + e \leq t + t' \\
&\leq t + t' + |\texttt{jmp}| \\
&\leq t + t' + |\texttt{jmp}| + \max(\pi') && \triangleleft \text{ Execution cost of the conditional} \\
&\leq t + \max(\pi_2) + e + \max(\pi_e) + |\texttt{jmp}| \\
&\leq t + \max(\pi_1) + e + \max(\pi_e) \\
&\leq o + o'
\end{aligned}$$

After this lemma it remains to prove that in the instrumentation, there is an execution which will output the cost t of the concrete semantics, thanks to the non-deterministic assignments.

Lemma 17. *Let s be a program, $\sigma^\sharp$ an alias abstract memory state, $\sigma \in \gamma_a(\sigma^\sharp)$ and s' the instrumentation of s: $(s', \_) = \mathbb{T}(s, \sigma_1^\sharp)$, then*

$$\forall h, t, u, o, \pi', \sigma' \begin{pmatrix} (s, \langle \sigma, \pi_\epsilon, h \rangle) \downarrow^t \langle \sigma', \pi', \_ \rangle \\ \wedge\ (s, \sigma, \sigma^\sharp) \Downarrow_{[u,o]} (\sigma', \_) \end{pmatrix} \Rightarrow \sigma_2[\mathsf{cost} \mapsto t] \in \mathbb{S}[\![s']\!]\sigma_1$$

Proof. The proof is also made by induction on the syntax of s. The instrumentation insert non-deterministic assignments to cost inductively on the syntax of s by respecting the range $[u, o]$ of the approximate semantics. Thanks to the previous lemma, we can select $t \in [u, o]$ the cost to execute the current statement, thus ensuring a correct final value for cost in a sequential semantics.

J.1 Block approximation

In this section we prove Lemma 15: the bounds computed by simulating the block with an alias analysis are sound.

Proof. Let $\mathsf{blk} = a_1; \ldots; a_n$, and $\sigma^\sharp$. The proof is made by recurrence on n. We rely again on a bi-simulation proof between the immediate semantics and the simulation semantics $[\![a]\!]_{\bowtie^\sharp}$ to keep an under or over-approximation.

The main issue of this proof is that the instruction in the simulated pipeline are not resolved: the simulation does not have access to the memory location pointed by registers. Thus it cannot tell if there is any data-dependencies between an instruction $\texttt{atom}'$ in the pipelines and the current one $\texttt{atom}$ being fetched. As stated in Section 4.1, the operators $\bowtie^\sharp_{\text{Must}}$ and $\bowtie^\sharp_{\text{May}}$ must satisfies the following two conditions.

$$\neg \bowtie^\sharp_{\text{Must}} (\texttt{atom}, \texttt{atom}', \sigma^\sharp) \implies \forall \sigma \in \gamma(\sigma^\sharp), \text{locks}(\texttt{atom}, \texttt{atom}', \sigma)$$

$$\bowtie^\sharp_{\text{May}} (\texttt{atom}, \texttt{atom}', \sigma^\sharp) \implies \forall \sigma \in \gamma(\sigma^\sharp), \neg\, \text{locks}(\texttt{atom}, \texttt{atom}', \sigma)$$

The operators first check the registers. Any registers that would be resolved (i.e. read) in $\texttt{atom}'$ is ignored and the WAW, RAW and WAR dependencies are checked. Once we have the guarantee that there is no data-dependencies between the registers, the alias analyses check that the content of these registers, if they are memory locations, may or must alias, depending on the bound being computed.

For the under-estimation, the condition on the must alias analysis is enough to ensure that the simulation will execute as much as or less cycles than the concrete semantics. If the simulation cannot fetched, i.e. if $\neg \bowtie^\sharp_{\text{Must}} (\texttt{atom}, \texttt{atom}', \sigma^\sharp)$, then there is a data-dependency between $\texttt{atom}$ and $\texttt{atom}'$ for all possible concrete states σ. So the concrete semantics cannot fetch either. Both execute cycle, and executing cycles preserve the bi-simulation relation. On the other hand, if the

must analysis does not detect any conflict, then the simulation will not execute cycle while the concrete semantics may actually do. In this case, the simulation ends up *late*, a notion formalized in Appendix J.2, with its proof of Lemma 1. The simulation has executed less cycles, but its pipeline contains more instructions than the concrete state, which may result in more conflicts later. The simulation can recover from this lateness but can never outrun it: even if it executes cycles while the concrete state do not, in total it will never execute more cycles than the concrete execution.

Similarly for the over-estimation, the condition on the may alias analysis is enough to ensure that the simulation will execute as much as or more cycles than the concrete semantics. Indeed, if the simulation does not detect a conflict and fetches the instruction, then the concrete semantics cannot be delayed either and will fetch the instruction too. However, the simulation may detect a conflict which does not exists in the concrete state. In that case, the simulation will execute a cycle while the concrete semantics does not, and so it is the concrete semantics which is *late*. In total, the concrete semantics cannot execute more cycle than the simulation from a may alias analysis.

J.2 Partial order on dependencies

The soundness of the approximated bound semantics heavily relies on the following lemma (adapted from Lemma 1).

Lemma 18. *Let* (σ, π, h) *be a processor state and let* s *be a program. If*

- s *is processed in* t *cycles from* (σ, π, h)*:* $(s, \sigma, \pi, h) \downarrow_t (\sigma', \pi', h')$
- *and it is processed in* t' *cycles from* (σ, π_ϵ)*:* $(s, \sigma, \pi_\epsilon, h) \downarrow_{t'} (\sigma', \pi'', h'')$

Then $t' \leq t$ *and* $t + \max(\pi') \leq \max(\pi) + t' + \max(\pi'')$

The proof of this lemma relies on the proof of preservation of the partial order by the directives and then by the big-step semantics. Then we use the fact that the lateness is a positive number to ensure the correct ordering of the costs.

Preservation of the Lateness Relation

Lemma 19. *Fetching an atomic instruction* `atom` *respects the following lockstep-simulation diagram. Black terms are hypothesis, blue ones are conclusions.*

$$\begin{array}{ccc} (\sigma, \pi_1) & \xrightarrow[\text{fetch } \texttt{atom}]{} & (\sigma', \pi_2) \\ \vdots \sqsubseteq_k & & \vdots \sqsubseteq_k \\ (\sigma, \pi_1') & \xrightarrow[\text{fetch } \texttt{atom}]{} & (\sigma', \pi_2') \end{array}$$

with $k \geq 0$

Proof. For all the variables written, the delay will be the same in π_2 and π_2': exactly $|\texttt{atom}|$. For the variables read, the delay is the maximum between the old delay and $|\texttt{atom}|$, the order between π_2 and π_2' is thus preserved. Now for the pipelines, we have four cases. First case, they take the same pipeline. This is the only delay in the p component that changes and it is set to 1 in both pipeline state. As $k \geq 0$ we preserve the order in that case. If π_1 chooses a pipeline X with a higher priority than the one, let say Y, chosen π_1', then we also respects the order. Indeed it means that the pipeline X was already occupied in π_1'. So $p_2(X) = 1 \leq p_1'(X) + k = p_2'(X) + k = 1 + k$ As for the pipeline Y its delay in π_2 is either 0 or 1 so is always bounded by $p_2'(Y) + k = k + 1$. Now if on the contrary X has a lower priority than Y, then the only explanation is that Y was already occupied in π_1. But Y was available in π_1', so $p_1(X) = 1 \leq p_1'(X) + k = k$. As the lateness k is at least of one cycles, the order is trivially respected for the pipelines: $\forall X \in \mathsf{Pips}, p_2(X) \leq p_2'(X) + k$.

Lemma 20. *Executing cycles to fetch an atomic instruction* `atom` *respects the following lockstep-simulation diagram. Black terms are hypothesis, blue ones are conclusions.*

$$\begin{array}{ccc} (\sigma, \pi_1) & \xhookrightarrow[\text{cycle } \mathit{atom}]{}{}^{t} & \pi_2 \\ \Big\vdots \sqsubseteq_k & & \Big\vdots \sqsubseteq_{k'} \\ (\sigma, \pi_1') & \xhookrightarrow[\text{cycle } \mathit{atom}]{}{}^{t'} & \pi_2' \end{array}$$

with $k' = k + t' - t \geq 0$.

Proof. The number of cycles t required by π_1 is the maximum latency of all resources (pipeline and variable) needed by `atom` in (σ, π_1). The same applies to π_1'. We need to show that t is less than $t' + k$, to ensure that $k' = k + t' - t \geq 0$. The order $\pi_1 \sqsubseteq_k \pi_1'$ implies that for any resources needed by `atom` in π_1, the delay d to get that resource is less than $d' + k$ where d' is the delay to get that same resource in π_1'. So the maximum of all delays in π_1, a.k.a t, is less than the $t' + k$, the maximum of all delays in π_1' plus k.

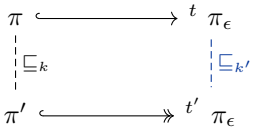
We now need to ensure that $\pi_2 \sqsubseteq_{k'} \pi_2'$. Let d_1 denotes the delay of any resource in π_1, that is either $w_1(v)$ or $r_1(v)$ for some $v \in \mathsf{Location}$ or $p_1(X)$ for some $X \in \mathsf{Pips}$. Let d_1' be the delay of the same resource but with respect to π_1', and d_2 and d_2' be the ones in π_2 and π_2' respectively. Executing cycles decrements these delay (without going below zero).

$$d_2 = max(0, d_1 - t) \text{ and } d_2' = max(0, d_1' - t')$$

$$
\begin{aligned}
d_2 &= max(0, d_1 - t) \\
&\leq max(0, d_1' + k - t) && \text{By } \pi_1 \sqsubseteq_k \pi_1' \\
&\leq max(0, d_1' + k - t + t' - t') \\
&\leq max(0, d_1' + k' - t') \\
&\leq max(0, d_1' - t') + k' && \text{Since } k' \geq 0 \\
&\leq d_2' + k'
\end{aligned}
$$

Which confirms that $\pi_3 \sqsubseteq_{k'} \pi_3'$.

Lemma 21. *Executing cycles to empty the pipelines respects the following lockstep-simulation diagram. Black terms are hypothesis, blue ones are conclusions.*

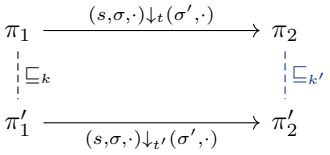


with $k' = k + t' - t \geq 0$.

Proof. Executing cycles to empty the pipelines is actually the same as executing cycles for an artificial instruction that would need all variables currently in the pipelines π and π' (we join these two sets of variables). So we can apply Lemma 20 to prove this lemma.

These two lemmas are enough to prove the preservation by the big-step semantics, by chaining the directives applied.

Lemma 22 (Lateness preservation). *Executing a statement s respects the following lockstep-simulation diagram. Black terms are hypothesis, blue ones are conclusions.*



with $k' = k + t' - t \geq 0$

Proof of approximated bounds In this section we prove the Lemma 1. Let π be a pipeline state, σ a variable state and s a program with s_0 the first instruction that will be fetched. We note s_1 the statement that will be executed after s_0 according to the semantics of s. Consider now a pipeline state π', having the same delay as π for every resource, except for the variables needed by s_0 where we impose a delay $n = \max(\pi)$.

$$\forall v \in \text{write}(s_0, \sigma), r'(v) = w'(v) = n \qquad \forall v \in \text{read}(s_0, \sigma), w'(v) = n$$

We have $\pi \sqsubseteq_0 \pi'$, as the delay we put in π' on the needed variables is necessarily greater than the delay in π.

We then execute the instruction s_0. Due to our artificial constraints, π' needs to wait exactly n cycles to be able to fetched it. Once it has executed these n cycles it is absolutely empty, thus the resulting state after the fetch of s_0, which we would note π'_2 is actually equal to the state $\pi_{\epsilon 2}$, that is π_ϵ after executing s_0.

We also have $\pi_\epsilon \sqsubseteq_0 \pi$ (no constraint on π_ϵ so all delays are equal to zero).

We can have the following diagram due to Lemma 22. On the first line, we have the cost starting from the empty pipelines π_ϵ, that is first 0 as there is no constraint and then t' by hypothesis. On the second line we have the cost starting from π which we decomposed into t_0 and t_1 such that $t = t_0 + t_1$ by hypothesis. Finally the third line corresponds to the cost starting from π' which first takes n steps as explained, and then the same as $\pi_{\epsilon 2}$ that is t'. The partial order delays are deduced from the Lemma 22.

$$\begin{array}{ccccc}
\pi_\epsilon & \xrightarrow{(s_0,\sigma,\cdot)\downarrow_0(\sigma',\cdot)} & \pi_{\epsilon 2} & \xrightarrow{(s_1,\sigma',\cdot)\downarrow_{t'}(\sigma'',\cdot)} & \pi'' \\
\vdots\sqsubseteq_0 & & \vdots\sqsubseteq_{t_0} & & \vdots\sqsubseteq_{t-t'} \\
\pi & \xrightarrow{(s_0,\sigma,\cdot)\downarrow_{t_0}(\sigma',\cdot)} & \pi_2 & \xrightarrow{(s_1,\sigma',\cdot)\downarrow_{t_1}(\sigma'',\cdot)} & \pi' \\
\vdots\sqsubseteq_0 & & \vdots\sqsubseteq_{n-t_0} & & \vdots\sqsubseteq_{n+t'-t} \\
\pi' & \xrightarrow{(s_0,\sigma,\cdot)\downarrow_n(\sigma',\cdot)} & \pi'_2 = \pi_{\epsilon 2} & \xrightarrow{(s_1,\sigma',\cdot)\downarrow_{t'}(\sigma'',\cdot)} & \pi''
\end{array}$$

With $t - t' \geq 0$ and $n + t' - t \geq 0$ which gives us the first bound $t' \leq t$.
Then we continue to empty the pipelines π' and π''.

$$\begin{array}{ccc}
\pi' & \xrightarrow{\quad\quad} & {}^{t_e}\pi_\epsilon \\
\vdots\sqsubseteq_{n+t'-t} & & \vdots\sqsubseteq_{n+t'-t+t'_e-t_e} \\
\pi'' & \xrightarrow{\quad\quad} & {}^{t'_e}\pi_\epsilon
\end{array}$$

where

$$\begin{aligned}
n + t' - t + t'_e - t_e &\geq 0 \\
\max(\pi) + t' - t + \max(\pi'') - \max(\pi') &\geq 0 \\
\max(\pi) + t' + \max(\pi'') &\geq t + \max(\pi')
\end{aligned}$$

We have the upper bound of the cost, which concludes the proof of Lemma 1.