Semantic Foundations for Cost Analysis of Pipeline-Optimized Programs

Gilles Barthe[1], Adrien Koutsos[2], Solène Mirliaz[3], David Pichardie[4], and Peter Schwabe[5]

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

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 [28,22], 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 [5] 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.

```
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.

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 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 [1].
- We have implemented our approach into Jasmin [3,4], 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 [3,4] 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 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

	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

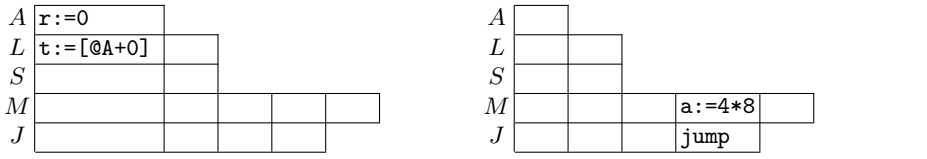
has a latency of 5, and is only supported by the pipeline M. This is a simple processor, real processors have more pipelines and can handle a larger instruction set. Note that the method presented in this paper is not specific to this processor: the number of pipelines, the instructions supported and 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*).



(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.

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 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;
```

```
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 $\texttt{atom} \in \mathsf{Atoms}$ 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.

Operands:

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

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

Labels:

ℓ	$\in \mathcal{L}$	

Statements:

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

Fig. 4: Syntax of the language

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 [3,4], 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.

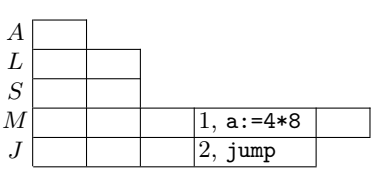
Memory State Values are stored at locations, $\mathsf{Location} = \mathsf{Reg} \cup \mathsf{MemLocs}$, comprising registers and memory locations. A memory state $\sigma : \mathsf{Location} \mapsto \mathsf{Val}$ is a map from locations to values, which are either integers or memory locations (see Fig. 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

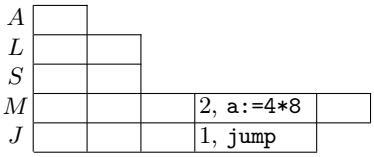
Latency		
$\lvert\texttt{atom}\rvert$	$\in \mathbb{N}$	
Values (Val) :		
v	$::= l \in \mathsf{MemLocs}$	Memory location
	$\mid n \in \mathbb{Z}$	Number
Locations ($\mathsf{Location}$):		
x	$::= l \in \mathsf{MemLocs}$	Memory location
	$\mid r \in \mathsf{Reg}$	Register
Memory state (S):		
σ	$\in \mathsf{Location} \to \mathsf{Val}$	
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

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 $\texttt{atom}$.



(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.

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 $\mathsf{Atoms_t}$. 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 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 ($(i, \mathtt{atom_t}) \in (\mathbb{N} \times \mathsf{Atoms_t})$), 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 $\pi(X_k) = (j, \mathtt{atom_t})$ with $j > i$) are replaced with ϵ. We illustrate this in Fig. 6, using the branch prediction example of Section 2. Recall that the 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

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

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

Fig. 7: Rules of data dependency locks

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: $\mathrm{BP\text{-}predict}(h, \ell)$ holds if the BP predicts that the jump at ℓ will be taken; and $h' = \mathrm{BP\text{-}update}(h, \ell, taken)$ 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.

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 (de-

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

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

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

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

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

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

fined in Fig. 7), which holds whenever the instruction $\mathtt{atom}$ *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 $\mathtt{atom}$. 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 $\mathtt{atom}$ 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.

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 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 $(s, \omega) \rightarrow^t (s', \omega')$ 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 the companion report [1].

Atomic The rules for $s_1 = \mathtt{atom}$ are ATOMIC and CYCLE. In the ATOMIC rule, we test whether the current state of the processor is ready to fetch atom using $\mathrm{ready}(\mathtt{atom}, \beta(\sigma), \pi)$. 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 $(i, \mathtt{atom})$ 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 $\mathtt{atom}; s$.

Conditional The rules SPEC-COND-TRUE-CORRECT and SPEC-COND-TRUE-INCORRECT define the behavior of the processor when encountering a conditional

$(s, \omega) \to^t (s', \omega')$
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

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-

$(s,\omega) \xrightarrow{=}{}^{t} (s',\omega')$
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

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

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

Fig. 11: Execution cost for small-step semantics

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 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 $(p;\texttt{skip},\omega) \to^{t} (\texttt{skip},\omega')$ 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

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
$\bowtie^\sharp_{\mathrm{May}}, \bowtie^\sharp_{\mathrm{Must}}$	$\in \mathsf{Atoms} \times \mathsf{Atoms} \times \mathsf{S}_a^\sharp \to \mathsf{bool}$	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

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 bounds $[u, o]$ of the cost to execute blk. The instrumentation relies on an alias

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 ($L = |\texttt{jmp}|$)

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 $[\![\cdot]\!]^\sharp_n$. We let R_0 be the input registers of our programs, and denote by $\iota^\sharp_n[s]$ 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:

$$\mathbb{C}(s) = \mathrm{proj}_{R_0 \cup \{\texttt{cost}\}}([\![s']\!]^\sharp_n(\iota^\sharp_n[s])) \quad \text{where} \quad (s', \_) = \mathbb{T}(s, \iota^\sharp_a[s]))$$

Block Instrumentation The block instrumentation computes the cost with $[\![\mathsf{blk}]\!]^\sharp$. It performs two simulations $[\![\mathsf{blk}]\!]_{\bowtie^\sharp_{\text{Must}}}$ and $[\![\mathsf{blk}]\!]_{\bowtie^\sharp_{\text{May}}}$ 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 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:

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

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*

$$(\sigma_0, \sigma_0) \in \gamma_n(\iota_n^\sharp[s]) \ \wedge \ \sigma_0 \in \gamma_a(\iota_a^\sharp[s])$$

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.

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

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

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. 14 and

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

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

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

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

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

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

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 the companion report [1], 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.

$$(\mathsf{blk}_1, \langle \sigma_1, \pi_\epsilon, h, \emptyset \rangle) \rightarrow^{t_1} \omega_1, \quad (\mathsf{blk}_2, \omega_1) \rightarrow^{t_2} (\mathtt{skip}, \omega_2) \text{ and } \omega_2 \hookrightarrow^{t'_2} \langle \sigma', \pi_\epsilon, h', \emptyset \rangle$$

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 $\pi_\epsilon, \emptyset$.*

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

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

The proof, given in the companion report [1], 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 $\mathbb{C}(s) = \mathrm{proj}_{R_0 \cup \{\mathsf{cost}\}}([\![s']\!]^\sharp_n(\iota^\sharp_n[s]))$ with $\mathbb{T}(s, \iota^\sharp_a[s]) = (s', \_)$. By Theorem 4, there exists o and u such that $(s, \sigma_0, \sigma^\sharp_0) \Downarrow_{[u,o]} (\sigma_1, \_)$. By Theorem 3, $\sigma_1[\mathsf{cost} \mapsto t] \in \mathbb{S}[\![s']\!](\sigma_0)$.

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

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

and in particular $\{\sigma_0\} \times \mathbb{S}[\![s']\!]\sigma_0 \subseteq \gamma_n([\![s']\!]^\sharp_n \iota^\sharp_n[s])$. 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 [3,4]. 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, 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 [6], 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)[6].

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

[6] 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.

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 [4] 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 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 [28], there has been a large body of work for analyzing the cost of programs using recurrence relations [2], program logics [25], type systems [26,21,14,23], 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* [24] 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 [30] 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 [27]. 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 [16,29,20]. 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. Companion report, https://hal.inria.fr/hal-03779257
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. J. Autom. Reason. **46**, 161–203 (2011)
3. 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)
4. 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)
5. Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B.: SoK: Computer-aided cryptography. In: Proc. of S&P 2021. pp. 777–795. IEEE (2021)
6. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.Y.: Easy-Crypt: A tutorial. In: Proc. of FOSAD. pp. 146–166. Springer (2013)
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. 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 (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. 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)
25. Nielson, H.R.: A Hoare-like proof system for analysing the computation time of programs. Sci. Comput. Program. **9**(2), 107–136 (1987)
26. Reistad, B., Gifford, D.K.: Static dependent costs for estimating execution time. In: Proc. of LFP'94. pp. 65–78. ACM (1994)
27. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Proc. of DATE'17. pp. 1697–1702. IEEE (2017)
28. Wegbreit, B.: Verifying program performance. J. ACM **23**(4), 691–699 (1976)
29. 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)
30. 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)