Modelling and proving the monotonicity of processor pipelines in Coq

Alban Gruin, Armelle Bonenfant, Thomas Carle and Christine Rochange
Institut de Recherche en Informatique (IRIT)
Université de Toulouse, CNRS, Toulouse INP, UT3, France
name.surname@irit.fr

***Abstract*—In critical real-time systems, the worst-case execution time (WCET) of software tasks must be statically bounded in order to guarantee that they all satisfy their timing constraints (e.g. deadlines). Obtaining such bounds is challenging due to the complexity of the software and of the acceleration mechanisms implemented in the hardware. Particular behaviors, known as timing anomalies, break some simplifying assumptions for the computation of WCET in single and multi-core processors. Some cores have been designed to implement a pipeline-level property known as monotonicity that guarantees that no timing anomaly can occur in the pipeline. Proving the monotonicity of a pipeline is tedious and error-prone, and so is reading the proof and convincing oneself of its validity. We thus propose to rely on the Coq proof assistant to guarantee the soundness of the proofs. In this paper, we show how the monotonicity property and the standard elements composing a pipeline can be modelled in Coq. Using an example based on an open-hardware RISC-V core from the literature, we introduce the main elements of the proof and discuss their reusability for other cores. We conclude that the model and proofs that we provide can be easily adapted to describe other in-order pipelines of equivalent complexity.**

***Index Terms*—Timing anomalies, Pipeline monotonicity, Coq proof assistant.**

I. Introduction

Real-Time systems are composed of increasingly complex programs running on increasingly complex hardware. Current processors feature mechanisms such as pipelines, caches, out-of-order and speculative execution of instructions, and embedded systems-on-chips include several of such cores, sometimes along with accelerators such as GPUs or vector processing units. Accurately modelling these components, their timing behavior and their interactions is a challenge from a static analysis perspective. Worst-Case Execution Time (WCET) analysis relies on such models in order to derive a safe but precise upper bound on the execution time of a program or task running on a particular hardware target. These analyses are sometimes unable to precisely determine the state of the hardware (e.g. the cache contents) at a particular point in the target program execution. In order to remain safe, the analyses must then consider all the combinations of all outcomes of uncertain events (e.g. cache hit or miss). To keep the analyses tractable, it is mandatory to make simplifying assumptions. However, these assumptions are invalidated by particular situations called timing anomalies (TAs). A TA occurs when a local worst-case situation (e.g. a cache miss) does not lead to the worst-case execution time of the program, or when the the difference between two possible outcomes of a local event gets amplified by the execution of the remainder of the program, to an unbounded extent. TAs were first characterized in out-of-order processors, but have since been shown to occur also in simple in-order pipelines [1]. In [2], [3], Hahn and Reineke propose a way to design TA-free pipelines and a modelling framework to prove the absence of TA in such pipelines. The proofs rely on the notion of monotonicity of the progress of instructions in the pipeline. In short, if each instruction is guaranteed to never be delayed by younger instructions in the pipeline (i.e. instructions that appear later in the program order), the progress of the instructions is monotonic. The monotonic property guarantees that in case of unpredictable event, the outcome that is locally slower always leads to longer a global execution time than the locally faster outcome. Moreover, this property also allows the derivation of safe upper-bounds on the global effect of a local event. The corresponding proof can be separated in two subsets: high-level, generic proof arguments that do not depend on a particular pipeline implementation, and pipeline specific proofs that rely on the pipeline topology and particularities.

In this paper, we provide elements to specify and prove the progress monotonicity property using the Coq proof assistant. Although our Coq proof targets the MINOTAuR pipeline [4], [5], we present the generic elements of the model and the proofs, so that the reader can use them to model and prove the monotonicity of other pipelines. The remainder of the paper is organised as follows: Sections II and III provide the formal definition of the pipeline model and of the monotonicity property. Section IV presents the proof. Then Section V presents the MINOTAuR core and discusses how the proof was applied to its pipeline. Section VI summarises the related work, and Section VII concludes the paper.

II. Abstract pipeline model

In this section, we give an overview of the pipeline model introduced in [2] and make it more generic (i.e. applicable to other pipelines than the SIC).

A program is seen as a sequence of instructions numbered in the order they enter the pipeline (i.e. the program order): $\mathcal{I} \coloneqq i_0, i_1, \ldots, i_n$. Each instruction has an opcode $opc(i)$

This work was partially supported by the ANR LabEx CIMI (grant ANR-11-LABX-0040) within the French State Programme "Investissements d'Avenir" and by the ANR ProTiPP project (grant ANR-22-CE25-0004).

2832-6520/24/$31.00 ©2024 IEEE
DOI 10.1109/MEMOCODE63347.2024.00007

that determines the nature of the functional unit that executes it.

A processor pipeline is described as a set of *stages* $\mathcal{S}$ partially ordered by relation $\sqsubset_{\mathcal{S}}$. Two virtual stages are added to $\mathcal{S}$: pre and $post$. According to $\sqsubset_{\mathcal{S}}$, pre (resp. $post$) is lower (resp. greater) than all the other stages of the pipeline, making the pipeline model a lattice.

At any given execution cycle, the *progress* of an instruction is in $\mathcal{P} \coloneqq \mathcal{S} \times \mathbb{N}$: it captures the stage the instruction is currently in, as well as the number of cycles (hereafter named *counter*) that the instruction must still spend in the stage. An order is defined on $\mathcal{P}$:

$$\forall (s_a,\, n_a), (s_b,\, n_b) \in \mathcal{P},$$
$$(s_a,\, n_a) \sqsubseteq_{\mathcal{P}} (s_b,\, n_b) :\Leftrightarrow s_a \sqsubset_{\mathcal{S}} s_b \vee (s_a = s_b \wedge n_a \geq n_b)$$

A *pipeline state* maps each instruction to its progress. The set of all possible pipeline states is $\mathcal{C} \subseteq \mathcal{I} \to \mathcal{P}$. Given two pipeline states c_a and c_b, c_b is said to have at least the progress of c_a if the progress of each instruction in c_b is superior or equal to its progress in c_a:

$$c_a \sqsubseteq c_b \Leftrightarrow \forall i \in \mathcal{I}\ .\ c_a(i) \sqsubseteq_{\mathcal{P}} c_b(i)$$

where $c(i)$ denotes the progress of instruction i in state c.

The behaviour of the pipeline, i.e. function $cycle(c)$ that maps a pipeline state c to its successor state, is specified through a set of functions that apply to c and determine how instructions will progress in the pipeline in the next cycle:

- $nstg(i)$ returns the next stage for instruction i (it depends on its current stage and its opcode);
- $lstg(o)$ returns the stage where opcode o completes its execution;
- $cnt(i)$ updates the counter of instruction i. If it is not zero, it is decremented by 1, reflecting that one cycle has passed;
- $next(i)$ returns if this instruction i is in pre and no other earlier instruction is in pre stage. That is to say, i is the next instruction being in pre stage which is ready to progress. $next(i)$ is generalised in $isnext(s,\, i)$ for any stage (not only pre);
- $pending(i)$ functions return booleans that indicate whether instruction i has control or data dependencies. In [2], three $pending$ functions are defined: $brpending$, $mempending$ and $stpending$ to deal with branch (resp. memory and store) operations. They are used to check whether the pipeline contains such operations that have not completed their execution yet. This family of pending functions can be generalised when adding an opcode as a parameter;
- $ready(i)$ returns a boolean that tells whether instruction i is ready to move to its next stage in the next cycle: it checks if its counter has reached 0, as well as additional, stage-specific rules (e.g. data dependencies are satisfied before entering a functional unit);
- $free(s)$ returns a boolean that tells whether a stage is able to accept a new instruction in the next cycle. This is true if the stage is currently empty, or if the instruction it contains will move to the next stage in the next cycle. The second part is computed recursively (will the next stage be able to accept an instruction?), also using the $ready$ function (is the instruction currently in the stage ready to move?).

In the rest of the document, these functions will be referenced as predicates.

All the program instructions are initially in the pre stage, and they all finish their execution in the $post$ stage. The program is completed once all instructions have reached the $post$ stage.

III. The monotonicity property

In [2], the monotonicity property is defined as follows:

Property 1. Pipeline monotonicity. *A pipeline meets the monotonicity property iff:*

$$\forall c_a,\, c_b \in \mathcal{C},\, c_a \sqsubseteq c_b \Rightarrow cycle(c_a) \sqsubseteq cycle(c_b)$$

If pipeline state c_b is more advanced than state c_a, the order is maintained when applying `cycle`.

Its demonstration is achieve thanks to this lemma:

Lemma 1. Update enable. *Let c_a, c_b two states $\in \mathcal{C}$. Let $i \in \mathcal{I}$ an instruction with equal progress in c_a and c_b ($c_a(i) = c_b(i)$) and each older instruction $j < i$ has progressed more in c_b than in c_a ($c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$). Then:*

$$\begin{cases} c_a.ready(i) \Rightarrow c_b.ready(i) \\ c_a.free(c_a.nstg(i)) \Rightarrow c_b.free(c_b.nstg(i)) \end{cases}$$

Lemma 1 expresses that an instruction cannot be delayed in its progress by a younger instruction. The proof is pipeline-dependent and relies on the enumeration of all possible cases for the progress of instruction i in c_a and c_b. Then the proof of the monotonicity property is generic: for any instruction i, since $c_a \sqsubseteq c_b$, either $c_a(i) \sqsubset_{\mathcal{P}} c_b(i)$ or $c_a(i) = c_b(i)$. In the former case, it is impossible for i to be more advanced in $cycle(c_a)$ than in $cycle(c_b)$ (at most $cycle(c_a)(i) = cycle(c_b)(i)$). In the latter case, the update enable lemma can be applied to conclude that if i advances to the next stage in c_a, then it does also in c_b: again, $cycle(c_a)(i) \sqsubseteq_{\mathcal{P}} cycle(c_b)(i)$.

Pen-and-paper proofs of the lemma and monotonicity are provided in [2], [3]. These are tedious, both to write and to read. They are also error-prone. This motivates our work towards an automatic (assisted) proof.

IV. General view of the Coq proof

The Coq proof assistant [6] is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. By using the Coq environment, we take advantage of the proof assistance process and improve our confidence in the proofs.

This section presents the generic elements of the Coq proof, that is elements that are pipeline-independent and can be used as a basis for any processor that can be described within the framework over-viewed in Section II.

A. Data structures

While the objective is to generate a Coq model that is as close as possible to the predicate-logic model of Section II, the perspective is a bit different: in the logic model, the focus is put on the pipeline stages and on the instructions they host; in the Coq version, the focus is on the instructions.

As shown in Listing 1, an `instruction` is represented by:

- an operation code (`opcode`: load, store, branch, etc.);
- an index `idx` that reflects the order in which instructions enter the pipeline, sometimes called instruction number;
- a list of initial latency valuations `latlist` for each pipeline stage;
- a boolean `pwrong` that indicates whether the instruction is mispredicted or not (used for branches).

The `progress` of an instruction is implemented as a couple: its current `stage` and its counter (remaining latency represented as a natural number).

The association of an `instruction` and its `progress` is paired in a couple: instruction progress (`instr_progress`).

The `pipeline_state` is then defined as a list of `instruction_progress`. This representation is generic, and can be enriched by adding metadata to the opcode, such as the operands of the instruction.

```
Definition latlist := list (stage * nat)%type.

Record instruction : Set := {
  opc : opcode ;
  idx : nat ;
  lats : latlist ;
  pwrong : bool
}.

Definition progress := (stage * nat)%type.
Definition instr_progress := (instruction * progress)%type.
Definition pipeline_state := list instr_progress.
```

Listing 1. Basic definitions in Coq. `stage` and `opkind`, being pipeline-specific, are not shown here.

The resulting data structure is straightforward to manipulate with Coq, although it requires demonstrating that the `cycle` function, which is subsequently defined, only modifies the progress of instructions in the pipeline, without altering the instructions themselves.

Comparators of instruction progress ($\sqsubseteq_{\mathcal{P}}$) and pipeline state ($\sqsubseteq$) are defined. In Listing 2, `stage_leb : stage -> stage -> bool` is the boolean implementation version of the relation order $\sqsubseteq_{\mathcal{S}}$ and `progress_leb` implements $\sqsubseteq_{\mathcal{P}}$.

Most comparators have a boolean equivalent that allows them to be used in functions, and an equivalence lemma to switch from one representation to the other one. The decidability of the stage equality is proved[1], and used to derive the boolean equality function, as shown in Listing 3.

As said in the introduction of this section, the representation differs from the original model introduced in Section II, where the set of instructions $\mathcal{I}$ is separated from the pipeline states. In the Coq model, a pipeline state holds each instruction and its progress. In terms of notations, in the remainder of the paper, ip refers to an instruction progress in a pipeline state and i to the actual instruction, characterized by its index. There is a direct equivalence in the notation between i in Coq, and i in the model of Section II.

[1]As the `stage` type should be a simple variant, the `decide equality` tactic is sufficient for this proof.

```
(* Definition of ⊑_S *)
Definition stage_leb s1 s2 :=
  match s1, s2 with
  | Pre, _ ⇒ true
  | ... (* depends on the pipeline model *)
  | Post, Post ⇒ true
  | Post, _ ⇒ false
  end.

(* Definition of ⊑_P *)
Definition progress_leb pra prb :=
  match pra, prb with
  | (stga, na), (stgb, nb) ⇒
     if stage_beq stga stgb then
       Nat.leb nb na
     else
       stage_leb stga stgb
  end.
```

Listing 2. Order relations in Coq

```
(* Decidable equality between stages *)
Lemma stage_eq_dec :
   forall (s s' : stage), {s = s'} + {s <> s'}.

(* Boolean definition of equality between stages *)
Definition stage_beq s s' :=
  if stage_eq_dec s s' then true
  else false.
```

Listing 3. Boolean stage equality in Coq

B. Predicates

Most of predicates and functions defined in the model are straightforward to transcribe in Coq. However, for some of them, it is convenient to demonstrate intermediate properties or lemma.

This can be illustrated by the `nstg` predicate that returns the next stage for an instruction progress. Its definition in the Coq model is associated with several basic properties that are straightforward to prove:

- the next stage is greater or equal to the current stage[2]: `stage_le (stg ip) (nstg ip)`;
- if the next stage is equal to the current stage, then they are both equal to `Post`;
- if the current stage is not equal to `Post`, then the next stage is strictly greater.

The definition of dependencies between instructions relies on predicates `depRaW` and `depWaW` (resp. for Read-after-Write and Write-after-Write data hazards). They are defined with a parametric function of type `instr_progress -> instr_progress -> bool`. These dependencies depend only on the opcode of the instructions, and not on the

[2]As mentioned earlier, `stage_leb` is the boolean function version of the `stage_le` property.

current progress of the instructions. Based on these definitions, the Coq model defines the `dep` function that determines whether an `instr_progress ip` depends on another `instr_progress jp` that is further away in the pipeline. This function heavily depends on the design of the pipeline.

Functions `pending`, `isnext`, and `ready` are implemented as described in Section II. They are straightforward to define. Their signature is shown in Listing 4.

```
pending : opcode → instr_progress →
      list instr_progress → bool

isnext : stage → instr_progress →
     list instr_progress → bool

ready : instr_progress → pipeline_state → bool
```

Listing 4. Basic predicate signatures

In particular:

- the model of Section II mentions three variant of the `pending` predicate, one for each type of instruction that is considered. In our Coq model, we define this predicate in a generic fashion: `pending` is parameterized with an opcode. This facilitates the proofs by treating all cases with a single predicate;
- `isnext` depends on the stage order defined earlier;
- `ready` is pipeline-specific and will be described in more details in Section V.

For the `free : stage -> pipeline_state -> bool` predicate, special care is required. As defined in the models we consider, it is recursive on its first argument (a pipeline stage s): if s hosts an instruction i that is ready to move to the next stage in the next cycle, then the stage is considered to be free if the next stage of i is free. This recursion extends to the `Post` stage, which is always free regardless of the pipeline structure.

It is necessary to have a well-founded relation on stages in order to stop the recursion in the definition of `free`. In Coq, it is obtained by using `Post` as a minimal element for the relation order $\sqsupset_{\mathcal{S}}$[3] and by defining `measure_stage : stage -> nat` that maps each stage to a number.

In practice, $\sqsubseteq_{\mathcal{S}}$ is reduced into `measure_stage` with the `<` (`Peano.lt`) order on natural numbers. Thus:

```
forall s s', stage_le s s' -> measure_stage s'
                < measure_stage s
```

with `Post` being valued to 0, and `Pre` having the highest valuation of all stages. The Coq model then shows that:

```
forall ip c, stg ip <> Post -> measure_stage
    (nstg ip c) < measure_stage (stg ip)
```

This proves `measure_stage` is indeed well-founded, and allows to define the `free` predicate.

Defining the `free` predicate as a `Fixpoint` has been experimented, but yielded hard-to-manipulate dependently-typed terms in the proof of Lemma 1. Hence, the `Fix` combinator is used instead, in the way shown in [7]. The Coq model shows that a de-recursified form of `free` is equivalent to its fully-recursive counterpart using the `Fix_eq` lemma. Depending on the pipeline mechanism, this proof may require the use of the functional extensionality axiom.

Finally, the `cycle` function, which maps each instruction progress of a given pipeline state to its next progress, is defined as shown in Listing 5, in which the `lat` and `cnt` operations are observers for the execution latency of an instruction progress in a given stage, and the counter of the current instruction progress in its current stage. This function is generic and does not depend on the pipeline structure.

```
Definition cycle_elt c ip :=
  let progress :=
    if ready ip c && free (nstg ip c) c then
      let nextstg := nstg ip c in
      (nextstg, lat ip nextstg)
    else
      (stg ip, (cnt ip) - 1) in
  (fst ip, progress).

Definition cycle c := map (cycle_elt c) c.
```

Listing 5. Implementation of the `cycle` function in Coq

C. Specification of hypotheses

In a pen-and-paper proof of the monotonicity of a pipeline, such as the one given in [2], some assumptions that are crucial to the validity of a proof are omitted. Even though for a human reader, they follow naturally from the system being modelled, they have to be clearly specified in the Coq model. For instance, in the pen-and-paper proofs, multiple pipeline states are considered, which associate each instruction in the sequence to an instruction state. Consequently, all pipeline states implicitly have the same size. In the chosen representation as a list of instruction progress, it is however necessary to explicitly state that for a same sequence of instructions, two pipeline states have the same length.

This section defines the major hypotheses that are used throughout the proof. Two of them (validity and consistency) are related to how a processor pipeline operates, and allow to discard cases that are impossible in practice. The last hypothesis is purely related to Coq and how our data structures are used in the proofs.

A pipeline state is considered valid if the current progress of each instruction is valid (no instruction progress corresponds to impossible cases).

A pipeline state is considered consistent with regard to another pipeline state if it is more advanced (the progress of each instruction in this state is greater or equal than in the other) and the order between the progress of each instruction is the same in both pipeline states. Consistency also requires other purely Coq-related conditions that are detailed in the remainder of the section. The consistency also checks minor properties such as the size of the pipeline state.

The way Coq deals with universality of variables, it is necessary to link variables together: for instance set that for all pipeline state designated by variable `ca` in the proofs, and for

[3]The pipeline may contain other stages that are always free (i.e. instructions are never stalled in these stages). If so, they also act as minimal elements.

all `instr_progress` designated by variable `ip`, `ip` is linked to `ca` by its localization in the instruction progress list.

We now provide a more in-depth presentation of these properties.

1) Validity: Listing 6 displays the definition of the validity property for the progress of an instruction. First, predicate `fits_in_stage` checks if the considered progress for the instruction corresponds to an impossible case: for example, an addition instruction cannot have a progress indicating that it resides in the Load/Store Unit (LSU). These cases are pipeline-specific. Additionally, the property ensures that the counter of the instruction does not exceed the initial value that this instruction gets assigned in its current stage.

```
Definition progress_valid e :=
  fits_in_stage (stg e) e ∧ cnt e <= lat (stg e) e.
```

Listing 6. Instruction validity property in Coq

Then, the validity of a pipeline state depends on the validity of the progress of each instruction, and on pipeline-specific conditions on stage capacities: in a valid state, no stage has more instructions than its capacity.

In the following proofs, all pipelines states that are manipulated are valid by hypothesis.

2) Consistent progress: In in-order pipelines, except in a few particular stages that may operate in parallel (e.g. execution stage with multiple parallel functional units) the instructions always respect the program order. The definition of the aforementioned `fits_in_stage` predicate contains the information on which stages allow instructions to progress outside of the program order. In the model, Property 1 is based on c_a and c_b such that $c_a \sqsubseteq c_b$, i.e. $\forall i \in \mathcal{I}$, $c_a(i) \sqsubseteq_{\mathcal{P}} c_b(i)$. However, this hypothesis that seems natural to a human reader is too weak in Coq for all stages in which instructions must progress in order. Indeed, with this hypothesis, it is possible to specify two pipeline states $c_a \sqsubseteq_{\mathcal{P}} c_b$ and two instructions i and j such that $c_a(i) \sqsupset_{\mathcal{P}} c_a(j) \wedge c_b(i) \sqsubset_{\mathcal{P}} c_b(j)$: j has less progress than i in c_a and has more in c_b. This means that the program order is not respected, either in c_a or in c_b. This in turn means that this definition of the hypothesis tolerates behaviors that are impossible in practice, and that preclude the proof of the monotonicity property. Although these impossible behaviors are easily discarded in a pen and paper proof, the Coq proof requires a stricter definition of the hypothesis. We restrict the definition using the `consistent_progress` predicate displayed in Listing 7. For all stages that maintain program order, if instruction i has less progress than j in c_a, then it must also have less progress than j in c_b, because necessarily $i > j$ in the program order. The `consistent_progress` predicate also checks that the instruction sequence has the same length in c_a and c_b and that the opcodes of i and j are consistent in both states. Finally, in processors that feature speculative execution, instructions can be flushed after a branch misprediction. Such instructions do not respect the program order, so they must be excluded from this restrictive hypothesis. This is done by not considering instructions j for which $c_b.stg(j) = post$. The speculative case is dealt with directly in the `ready` predicate, using the `pwrong` attribute of the corresponding instruction.

```
Definition consistent_progress (ca cb : pipeline_state) :=
  length ca = length cb ∧
  forall i i', In (i, i') (combine ca cb) →
    opc i = opc i' ∧ progress_le (snd i) (snd i') ∧
    (forall j j', In (j, j') (combine ca cb) →
              stg j' <> Post →
              stage_lt (stg i') (stg j') →
              fits_in_stage (stg i') j' →
              stage_le (stg i) (stg j) ∧ idx j < idx i).
```

Listing 7. Restrictions and consistency between c_a and c_b in Coq

The hypothesis defined in Listing 7 is sufficient to enable the proof of Lemma 1. Since it does not interfere with speculative execution, and that pipeline-specific cases are excluded with predicate `fits_in_stage`, this definition should work for all in-order pipelines.

3) Locality: The proof of monotonicity holds for all pipeline states. When an instruction progress is introduced in the hypothesis, it needs to be logically linked to a pipeline state. This logical link is the index of the instruction in the instructions list, designated by variable n in the proofs. In Listing 8, a series of variables and hypotheses stipulate that ip (resp. ip') corresponds to the progress of the instruction of index n in c_a (resp. c_b).

D. Proof strategy

The main goal of the proof is to show that a specific pipeline is monotonic, which means showing that: $\forall i \in \mathcal{I}$, $c_a(i) \sqsubseteq_{\mathcal{P}} c_b(i) \Rightarrow \forall i \in \mathcal{I}$, $cycle(c_a)(i) \sqsubseteq_{\mathcal{P}} cycle(c_b)(i)$. To do so, we assume any instruction i such that $c_a(i) \sqsubseteq_{\mathcal{P}} c_b(i)$, and show that the order is preserved by applying the cycle function to both c_a and c_b.

The proof strategy is schematized in Figure 1 by an extract of the tree of proof dependencies. The root `is_monotonic` theorem is proven using two lemmas `unmoved_is_monotonic` (in Subsection IV-D2) and `moved_is_monotonic` (in Subsection IV-D1). In this figure, consistent refers to the consistent hypothesis; pending, free, ready, isnext refer to their corresponding predicates; flush refers to the pipeline flush mechanism; remains refers to conservation across cycle. Generic properties and lemmas are depicted in green: they can be reused for any in-order pipeline. The properties depicted in blue are pipeline-specific.

The main theorem `is_monotonic` is detailed in Listing 8. The theorem can be read as follows: for two valid pipeline states c_a and c_b, with c_b being consistently more advanced than c_a, let ip be an instruction progress in c_a, ip' be the instruction progress of the same instruction i in c_b, if progress ip is lower ($\sqsubseteq_{\mathcal{P}}$) than ip' then $cycle(c_a)(i) \sqsubseteq_{\mathcal{P}} cycle(c_b)(i)$.

At the highest level, the proof works by dissociating two cases: either $c_a(i) \sqsubset_{\mathcal{P}} c_b(i)$ or $c_a(i) = c_b(i)$. In the remainder of the section, we provide the main ideas behind the proof of both cases. The proof of the first case is completely generic and works for any in-order pipeline. The equality

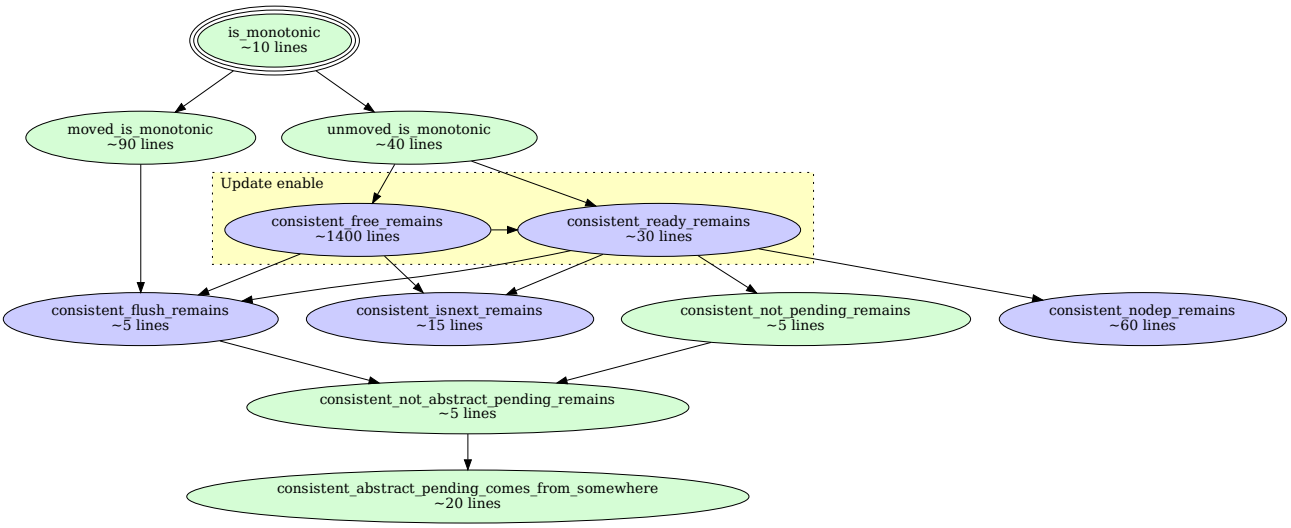

Fig. 1. Proof dependencies

```
Variable ca cb ca' cb' : pipeline_state.
Variable ip ip' : instr_progress.
Variable stga stgb : stage.
Variable o : ocpode.

(* Validity *)
Hypothesis Hvalida : pipeline_valid ca.
Hypothesis Hvalidb : pipeline_valid cb.

(* Consistency *)
Hypothesis Hconsistent : consistent_progress ca cb.

(* Locality *)
Variable n : nat.
Variable capre cbpre capost cbpost: pipeline_state.
Hypothesis Hca : ca = capre ++ ip :: capost.
Hypothesis Hcb : cb = cbpre ++ ip' :: cbpost.
Hypothesis Hn : n = List.length capre.
Hypothesis Hn' : n = List.length cbpre.

(* Monotonicity *)
Hypothesis Hca' : ca' = cycle ca.
Hypothesis Hcb' : cb' = cycle cb.

Theorem is_monotonic : forall na nb,
 ip = (o, (stga, na)) → ip' = (o, (stgb, nb)) →
 progress_leb (stga, na) (stgb, nb) = true →
 option_progress_leb (nth_error ca' n) (nth_error cb' n) =
     ↪ true.
```

Listing 8. Monotonicity in Coq

case, however, is proven using a variant of the update enable property (Lemma 1), which is pipeline-specific.

1) $c_a(i) \sqsubset_{\mathcal{P}} c_b(i)$ *(Coq lemma* `moved_is_monotonic`*):* Either i is in a less advanced stage in c_a than in c_b, or it is in the same stage, but has a higher counter in c_a. In the first case, whatever happens, i can at most reach the more advanced stage in $cycle(c_a)$, but in this case, it has a higher counter than in $cycle(c_b)$, or i has also moved to another stage in $cycle(c_b)$. In the second case, the counter of i is decremented by one in $cycle(c_a)$ which means that it is either higher or equal to the counter in $cycle(c_b)$ or that i moves to another stage in $cycle(c_b)$.

This is summarized in the next two lemmas, that translate in Coq to the goals of Listing 9.

Lemma 2. *Different stages. When* $c_a.stg(i) \sqsubset_{\mathcal{S}} c_b.stg(i)$*, then* $cycle(c_a)(i) \sqsubseteq_{\mathcal{P}} cycle(c_b)(i)$.

Lemma 3. *Equal stages. When* $c_a.stg(i) = c_b.stg(i)$*, then* $cycle(c_a)(i) \sqsubseteq_{\mathcal{P}} cycle(c_b)(i)$.

```
(* Moved is monotonic *)
option_progress_leb
  (Some (fst ip, if ready ip ca && free (nstg ip ca) ca
                 then (nstg ip ca, lat ip (nstg ip ca))
                 else (stg ip, cnt ip - 1)))
  (Some (fst ip', if ready ip' cb && free (nstg ip' cb) cb
                  then (nstg ip' cb, lat ip' (nstg ip' cb))
                  else (stg ip', cnt ip' - 1))) = true
```

Listing 9. Goal to prove when `ip` is not equal to `ip'`

2) $c_a(i) = c_b(i)$ *(Coq lemma* `unmoved_is_monotonic`*):* In this case, either i is not completely processed in its current stage (in both c_a and c_b), and in this case its counter is decremented in the next cycle in both states, or it is completely processed. In this case, we need to show that if i goes to the next stage in the next cycle in c_a, it does so also in c_b.

```
option_progress_leb
  (Some (fst ip, if ready ip ca && free (nstg ip ca)
                 then (nstg ip ca, lat e (nstg ip ca)
                 else (stg ip, cnt ip - 1))))
  (Some (fst ip, if ready ip cb && free (nstg ip cb)
                 then (nstg ip cb, lat e (nstg ip cb)
                 else (stg ip, cnt ip - 1)))) = true
```

Listing 10. Goal to prove when `ip` is equal to `ip'`

That case is proven by using the goal shown in Listing 10, which is equivalent to:

$$c_a.ready(i) \wedge c_a.free(c_a.nstg(i)) \Rightarrow \\ c_b.ready(i) \wedge c_b.free(c_b.nstg(i))$$

This resembles the update enable property, but allows correcting a contradiction that can arise in the presence of stages that are able to host multiple instructions. Given a state c_a, with instructions h and i such that:

$$\begin{cases} c_a.stg(h) = c_a.stg(i) \\ c_a.nstg(h) = c_a.nstg(i) \\ c_a.ready(h) \wedge \neg c_a.ready(i) \\ c_a.free(c_a.nstg(h)) \end{cases}$$

it is possible to construct c_b such that:

$$\begin{cases} c_b.stg(h) = c_a.nstg(h) \\ \neg c_b.ready(h) \end{cases}$$

which means that $\neg c_b.free(c_b.nstg(i))$. Changing the definition of the update enable property to assume $c_a.ready(i)$ alongside $c_a.free(i)$ removes this contradiction. Additionally, it is generally possible to demonstrate that $c_a.ready(i) \Rightarrow c_b.ready(i)$, meaning that if an instruction progress is ready in c_a, it remains ready in c_b.

The proof of this property is detailed in Section V as it is pipeline-specific.

Previous subsections all together allow to define a general approach for implementing our pipeline model in Coq. So far, no pipeline-specific mechanisms were used for the proofs.

V. Pipeline-specific proofs

We illustrate the pipeline-specific portion of the proof using the MINOTAuR core. Like the SIC, this processor was previously proven to be free of timing anomalies, using pen and paper proofs. We choose this processor to showcase our Coq proof, as it is more recent and a bit more complex than the SIC. A model of the core is presented in Figure 2. It is based on the CVA6 RISC-V core [8], which is a 6 stages issue-in-order core that comprises various advanced features. The execution unit is composed of several functional units (ALU, CSR unit, integer multiplication and division units and a Load/Store Unit with separate Load and Store stages) that can operate in parallel. As a consequence, a scoreboard is present to deal with dependencies between instructions and a reorder buffer tackles exception-related issues. Using these components, instructions are reordered and committed in the program order (when they exit the CO stage). To improve the average-case performance, the core features instruction queues and a store buffer. Finally, three branch predictors (a Branch History Table, a Branch Target Buffer and a Return Address Stack) support speculative execution.

A. Pipeline-specific model instantiation

We do not display the complete model of the core here as it is not an original contribution of this paper. However, since the pipeline-specific parts of our proofs deal with the

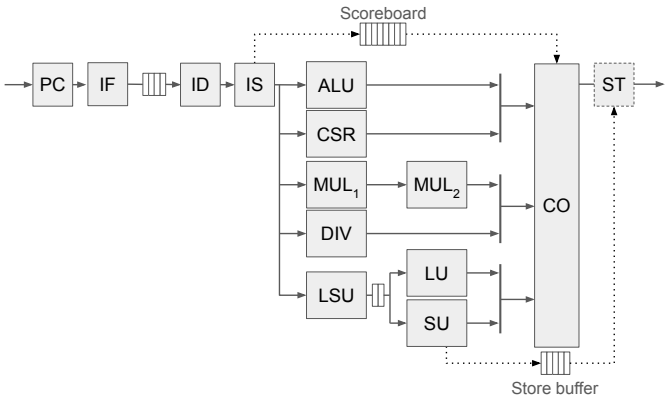

Fig. 2. Model of the MINOTAuR core adapted from [5]

ready and *free* predicates, we detail their formal definition in Figure 3 for illustration purposes. In the *ready* predicate, the first term states that a speculative instruction is ready to be flushed when a mispredicted branch is resolved. In all other cases, the instruction must have its counter equal to 0. Additionally, three terms model how instructions can be blocked in the pipeline. First, instructions can be blocked in the PC computation (PC) stage in order to prevent conflicts on the memory bus that would make the pipeline non-monotonic. Second, instructions can be blocked in the issue (IS) stage if an unresolved dependency exists with another instruction or if a CSR instruction is pending and the current instruction is not a memory instruction. Also, to prevent conflicts on the shared result bus between the division and multiplication units (which would make the pipeline non-monotonic), corresponding instructions are stalled in the issue stage as long as a division is being processed. Finally, load, store and atomic instructions are stalled in the LSU when an atomic instruction is pending, and loads are also stalled as long as the store buffer is not empty to prevent conflicts on the memory bus that would jeopardize the monotonicity of the pipeline.

We have encoded this model in Coq following the approach described in Section IV. We now describe the philosophy behind the pipeline-specific proof of our variant of the update enable property that we left unspecified at the end of the previous section.

We need to prove that for $c_a \sqsubseteq c_b$, $c_a(i) = c_b(i)$:

$$c_a.ready(i) \wedge c_a.free(c_a.nstg(i)) \Rightarrow \\ c_b.ready(i) \wedge c_b.free(c_b.nstg(i))$$

To do so, we start by proving two lemmas that respectively state that: $c_a.ready(i) \Rightarrow c_b.ready(i)$ and that $c_a.ready(i) \wedge c_a.free(c_a.nstg(i)) \Rightarrow c_b.free(c_b.nstg(i))$. Then the proof proceeds straightforwardly by applying the lemmas.

Lemma 4. *Ready remains.* $\forall i \in \mathcal{I}, c_a.ready(i) \Rightarrow c_b.ready(i)$

Proof. Depending on the considered stage and type of instruction, `ready` evaluates the `pending`, `dep` and `isnext`

$$
\begin{aligned}
&c.ready(i) \coloneqq (pwrong(i) \wedge c.flush(i)) \vee (c.cnt(i) = 0 \wedge c.isnext(c.stg(i),\, i)) \\
&\qquad \wedge (c.stg(i) = \text{PC} \Rightarrow \neg c.pending(i,\, branch) \wedge (ichit(i) \\
&\qquad\quad \vee (\neg c.pending(i,\, load) \wedge \neg c.pending(i,\, store) \wedge \neg c.pending(i,\, atomic)))) \\
&\qquad \wedge (c.stg(i) = \text{IS} \Rightarrow (opc(i) \notin \{load,\, store,\, atomic\} \Rightarrow \neg c.pending(i,\, csr)) \\
&\qquad\quad \wedge (opc(i) \in \{mul,\, div\} \Rightarrow \neg c.pending(i,\, div)) \\
&\qquad\quad \wedge (\forall j < i \,.\, (dep_{RaW}(i,\, j) \Rightarrow (lstg(opc(j)),\, 0) \sqsubseteq_{\mathcal{P}} c(j) \wedge (opc(j) = csr \Rightarrow c.stg(j) \sqsupset_{\mathcal{S}} \text{CO})) \\
&\qquad\quad\quad \wedge (dep_{WaW}(i,\, j) \Rightarrow c.stg(j) \sqsupset_{\mathcal{S}} \text{CO} \vee (c.isnext(\text{CO},\, j) \wedge opc(j) \neq csr)))) \\
&\qquad \wedge (c.stg(i) = \text{LSU} \Rightarrow (opc(i) \in \{store,\, atomic\} \wedge \neg c.pending(i,\, atomic)) \\
&\qquad\quad \vee (opc(i) = load \wedge (\neg c.pending(i,\, store) \wedge \neg c.pending(i,\, atomic))))
\end{aligned}
$$

$$
\begin{aligned}
&c.free(s) \coloneqq s \in \{\text{ALU},\, \text{MUL}_1,\, \text{CSR},\, \text{MUL}_2,\, \text{CO},\, post\} \\
&\qquad \vee (s \in \{\text{IF},\, \text{IS},\, \text{LSU},\, \text{SU}\} \wedge c.slot(s)) \\
&\qquad \vee (s \in \{\text{PC},\, \text{ID},\, \text{DIV},\, \text{LU},\, \text{ST}\} \wedge ((\neg \exists j \,.\, c.stg(j) = s) \vee (\exists j \,.\, c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j))))) \\
&\qquad \vee (\exists i \,.\, c.stg(i) = s \wedge pwrong(i) \wedge \neg c.pending(i,\, branch))
\end{aligned}
$$

Fig. 3. Definition of the *ready* and *free* predicates of the MINOTAuR core.

predicates (in inverted logic for the first 2). It must then be shown that the valuation of these predicates is the same in c_b and in c_a, in all cases where $c_a.ready(i)$ is true. In Coq, this is done on a case by case basis.

For `pending`, the Coq goal amounts to: `forall opcode i i', pending opcode i ca = false -> pending opcode i' cb = false`. The proof is based on the fact that $c_a.ready(i)$ is true if `pending` is false. We thus have to prove that `pending` is false in c_b when it is false in c_a. The proof works by contraposition. If there exists a pending instruction j (depending on the case, j can be a branch, a load, a store, an atomic or a csr) in c_b, then as $c_a \sqsubseteq c_b$, j is also pending in c_a and thus i is not ready in c_a.

For `dep`, we proceed with the same philosophy: an unresolved dependency (RaW hazard) for instruction i in c_b means that an instruction j has not yet been entirely processed. Since $c_a \sqsubseteq c_b$, j cannot be entirely processed in c_a either. Thus if i is not ready in c_b because of an unresolved dependency, it is not ready in c_a either. Other hazards (WaW hazards) are dealt with in the same fashion.

Finally, for `isnext` it must be shown that `forall stg i i', isnext stg i ca = true -> isnext stg i' cb = true`.

If $c_a.isnext(c_a.stg(i),\, i)$ then all older instructions $j < i$ are already in a more advanced stage in c_a: $c_a.stg(i) \sqsubseteq_{\mathcal{S}} c_a.stg(j)$. Since $c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$, we have $c_a.stg(j) \sqsubseteq_{\mathcal{S}} c_b.stg(j)$, so by transitivity, $c_a.stg(i) \sqsubset_{\mathcal{S}} c_b.stg(j)$. Now, the main assumption here is that $c_a(i) = c_b(i)$ so the proof concludes that $c_b.stg(i) \sqsubset_{\mathcal{S}} c_b.stg(j)$.

As `ready` is a conjunction of `isnext`, `pending` and other dependencies and hazard checks, it is possible to conclude that `forall i, ready i ca = true -> ready i cb = true`. The exact nature of the condition can vary depending on the stage of i, requiring a case-by-case proof. Using the Coq proof assistant ensures that no case is forgotten. □

Lemma 5. *Free remains.* $\forall i \in \mathcal{I},\, c_a.ready(i) \wedge c_a.free(c_a.nstg(i)) \Rightarrow c_b.free(c_b.nstg(i))$

Proof. If $c_a.free(s)$ is true, it means that one of the following statements is true:

- the stage hosts fewer instructions than its maximal capacity;
- the stage hosts as much instructions that it can, but one of them is ready, and its next stage is also free.

Most stages (PC, ID, ...) have a capacity of one instruction. For those stages, proving the lemma is equivalent to showing that both following statements are true:

$$
\begin{aligned}
&(\nexists j \,.\, c_a.stg(j) = c_a.nstg(i)) \Rightarrow \\
&(\exists j \,.\, c_b.stg(j) = c_b.nstg(i) \wedge c_b.ready(j) \wedge \\
&\qquad c_b.free(c_b.nstg(j))) \vee \\
&(\nexists j \,.\, c_b.stg(j) = c_b.nstg(i))
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
&(\exists j \,.\, c_a.stg(j) = c_a.nstg(i) \wedge c_a.ready(j) \wedge \\
&\qquad c_a.free(c_a.nstg(j))) \Rightarrow \\
&(\exists j \,.\, c_b.stg(j) = c_b.nstg(i) \wedge c_b.ready(j) \wedge \\
&\qquad c_b.free(c_b.nstg(j))) \vee \\
&(\nexists j \,.\, c_b.stg(j) = c_b.nstg(i))
\end{aligned}
\tag{2}
$$

To prove Statement 1, it is demonstrated that if $\nexists j \,.\, c_a.stg(j) = c_a.nstg(i)$, then $\nexists j \,.\, c_b.stg(j) = c_b.nstg(i)$. This can be shown by contraposition: if $\exists j \,.\, c_b.stg(j) = c_b.nstg(i)$, then, by construction, $c_a.stg(j) = c_a.stg(i) \vee c_a.stg(j) = c_a.nstg(i)$. In the former case, i cannot be ready because it is not the next instruction in the stage, or cannot be in its stage because there would be two instructions in a single stage, depending on the structure of the pipeline. It can be concluded that $c_a.stg(j) = c_b.stg(j)$.

For Statement 2, there are two cases: one where $c_a(j) = c_b(j)$, one where $c_a(j) \sqsubset_{\mathcal{P}} c_b(j)$. In the latter case, as j left the stage, there cannot be another instruction k, otherwise $c_a.stg(k) = c_b.stg(k) = c_a.stg(j)$ (as shown earlier), meaning that there would be two instructions in the stage, which is not possible. Hence, $c_b.free(c_b.nstg(i))$ is true, because $c_b.nstg(i)$ is empty. This proof relies on the trace validity hypothesis, in which it is considered that a stage cannot host more instructions that its capacity. When $c_a(j) = c_b(j)$, the instruction is still ready, as proven in Lemma 4.

This argument is extended in the case of stages capable of handling multiple instructions. The number of instructions hosted by a stage in c_a is higher or equal than in c_b. Otherwise, an instruction would have overtaken i. This is not possible because of the consistency property.

The last part of the proof relies on the demonstration that the property that is being demonstrated for $c_a.stg(i)$, was demonstrated for each stage reachable from that stage. □

To bring the proof to completion, we chose to prove it, one lemma for each stage, starting from the first one that is not always free. The proofs can be simplified by using a custom tactic.

B. Impact of speculative execution on the proof

To accommodate the model of MINOTAuR to speculative execution, new predicates were added to know if an instruction is mispredicted ($pwrong(i)$), and if it should be flushed in the next cycle ($c.flush(i)$).

Some predicates defined earlier are modified to have the following properties:

$$pwrong(i) \wedge c.flush(i) \Rightarrow \begin{cases} c.ready(i) \\ c.free(c.stg(i)) \\ c.nstg(i) = post \end{cases}$$

This has an impact on the proofs. On the update enable property, it must be shown that:

- an instruction that is ready in c_a because it is flushed is also ready in c_b, i.e. $c_a.flush(i) \Rightarrow c_b.ready(i)$. This impacts Lemma 4.
- a stage that is free in c_a because one of its instructions is flushed is also free in c_b, i.e. $c_a.flush(i) \Rightarrow c_b.free(c_b.stg(i))$. This affects Lemma 5.

We detail the impact on both lemmas later in this section, but we must first introduce how the `flush` function is handled.

The `flush` function is similar to `pending`: the latter checks if there is an instruction of a certain opcode older than i up to a certain stage, while the former checks if there is a control flow instruction that is not mispredicted up until the ALU. It is therefore possible to abstract the `pending` function so that it tests if a boolean function taking an instruction is satisfied by any instruction between `i` and a specific stage, as shown on Listing 11.

```
Definition abstract_pending cond stg i c :=
  existsb (fun i' ⇒ (idx i' <? idx i) && cond i' &&
                    progress_ltb (snd i') (stg, 0)) c.

Definition pending opcm := abstract_pending
  (fun i' ⇒ opcode_beq (opc i') opcm) (lstg opcm).

Definition spec := abstract_pending
  (fun i' ⇒ negb (pwrong (fst i')) &&
            opcode_beq (opc i') Branch) ALU.

Definition flush i c :=
  negb (spec i c).
```

Listing 11. Goal to prove when `i` is not equal to `i'`

Next, it is possible to show the following property, using a strategy similar to the one described in Lemma 4:

```
forall cond stg i i',
abstract_pending cond stg i ca = false ->
 abstract_pending cond stg i' cb = false
```

With this abstract `flush` function, we can now proceed with the modifications to the proofs of Lemma 4 and 5.

1) Changes on Lemma 4: It is now necessary to take two cases into account:

- when $pwrong(i) \wedge c_b.flush(i)$, then $c_b.ready(i)$, independently on the valuation of $c_a.ready(i)$;
- when i is not flushed in c_a and c_b, the rest of Lemma 4 is left unchanged.

A similar argument can also be made for Lemmas 2 and 3: when $pwrong(i) \wedge c_b.flush(i)$, then $cycle(c_b)(i) = (post, 0)$, which is the highest progress possible: $cycle(c_a)(i) \sqsubseteq_{\mathcal{P}} cycle(c_b)(i)$.

2) Changes on Lemma 5: Since $\forall c, c.flush(i) \Rightarrow c.free(c.stg(i))$, it follows that if $\forall i, (\exists j \ . \ c_a.stg(j) = c_a.nstg(i) \wedge pwrong(j) \wedge c_a.flush(j)) \Rightarrow c_a.free(c_a.nstg(i))$. Two cases are possible:

- if $c_a.stg(j) = c_b.stg(j)$, as $c_a.flush(j) \Rightarrow c_b.flush(j)$, it follows that $c_b.free(c_b.stg(j))$;
- if $c_a.stg(j) \sqsubset_{\mathcal{S}} c_b.stg(j)$, then there is no instruction in $c_b.nstg(i)$ in c_b, hence $c_b.free(c_b.nstg(i))$.

These changes, once implemented in lemmas `unmoved_is_monotonic` and `moved_is_monotonic`, allow to prove the monotonicity of the modelled pipeline. The main theorem, `is_monotonic`, use both lemmas after enumerating the two possible configurations for `ip` and `ip'`.

VI. Related work

Timing predictability is a fundamental property for the verification and validation of critical systems. With regard to execution time analysis of applications running on multicore processors, the community agrees on the relevance of favoring execution cores that are free of timing anomalies [9], [10]: they allow adopting a compositional approach [11] and considering individual thread execution times to build up the overall execution time of the application.

In [12], the monotonicity of the pipeline's behaviour is shown to be a sufficient condition to prevent timing anomalies. As already mentioned, the same authors have designed a monotonic processor pipeline (named SIC), provided a formal model of it and used this formal model to prove monotonicity by hand [2]. This work was taken up in [4], whose authors propose their own monotonic processor (MINOTAuR) and prove its monotonicity using the same framework. Their pen-and-paper proofs are long and difficult to read, which gave us the idea of using the Coq proof assistant.

Formal methods can help in verifying properties on hardware architectures. In [13], the Coq proof assistant is used to assess the correctness of the implementation of complex instructions through microprograms. Kami [14], a Coq library,

enables the specification and verification of multicore architectures with coherent cache memories. In [15], the specification in Haskell of a RISC-V processors is interfaced with several tools, among which Coq to verify functional properties. A Coq-based framework to design and verify memory controllers is proposed in [16]. In [17], the TriCore architecture is modelled with UCLID5; model checking and SMT-solving are used to detect the possible occurence of timing anomalies. This approach differs from ours: it searches for situations in which a timing anomaly could be observed in a non-predictable processor, while we instead prove the absence of timing anomalies for a processor that has been designed to prevent them. In [3], the monotonicity and compositionality of the SIC processor mentioned above are shown using an SMT formulation. However, the authors report non-scalable solving times.

VII. Conclusion

We propose a strategy to prove the monotonicity of processor pipelines using the Coq proof assistant. Our strategy comprises a pipeline-independent portion that can be used as is for any in-order pipeline, and a pipeline-dependent part that instantiates the model of a particular pipeline and enumerates cases on this particular instance. In order to demonstrate the validity of our approach, we model the MINOTAuR core in the Coq language and perform the pipeline-dependent proofs on this model. As MINOTAuR features advanced and complex execution mechanisms (instruction queues, parallel functional units, scoreboard, speculative execution), being able to prove its monotonicity demonstrates the versatility of our approach for in-order pipelines. In the future, we plan to work on two complementary directions: (i) automating the pipeline-dependent portions of the proofs as much as possible, for example by automatically generating parts of the Coq model and proofs, and (ii) working on an even more general pipeline model to obtain a completely pipeline-independent proof, then each particular pipeline must only be proven to respect the assumptions of the general model.

References

[1] S. Hahn, M. Jacobs, and J. Reineke, "Enabling compositionality for multicore timing analysis," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS '16*, 2016.

[2] S. Hahn and J. Reineke, "Design and analysis of SIC: A provably timing-predictable pipelined processor core," in *IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 469–481.

[3] ——, "Design and analysis of SIC: A provably timing-predictable pipelined processor core," *Real Time Systems*, vol. 56, no. 2, pp. 207–245, 2020.

[4] A. Gruin, T. Carle, H. Cassé, and C. Rochange, "Speculative execution and timing predictability in an open source RISC-V core," in *IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 393–404.

[5] A. Gruin, T. Carle, C. Rochange, H. Cassé, and P. Sainrat, "MINOTAuR: A timing predictable RISC-V core featuring speculative execution," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 183–195, 2022.

[6] The Coq Development Team, "The Coq Proof Assistant." [Online]. Available: https://doi.org/10.5281/zenodo.1003420

[7] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[8] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[9] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A Definition and Classification of Timing Anomalies," in *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, ser. OpenAccess Series in Informatics (OASIcs), F. Mueller, Ed., vol. 4. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/671

[10] B. Binder, M. Asavoae, B. Ben Hedia, F. Brandner, and M. Jan, "Is this still normal? putting definitions of timing anomalies to the test," in *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2021, Houston, TX, USA, August 18-20, 2021*, 2021, pp. 139–148.

[11] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: definition and challenges," *ACM SIGBED Review*, vol. 12, no. 1, pp. 28–36, 2015.

[12] ——, "Toward compact abstractions for processor pipelines," in *Correct System Design*. Springer, 2015, pp. 205–220.

[13] L. Arditi, "Formal verification of microprocessors: a rst experiment with the Coq proof assistant," Université de Nice, France, Tech. Rep., 1996.

[14] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Programmming Languages*, Aug. 2017.

[15] T. Bourgeat, I. Clester, A. Erbsen, S. Gruetter, A. Wright, and A. Chlipala, "A multipurpose formal RISC-V specification," *arXiv preprint arXiv:2104.00762*, 2021.

[16] F. L. Malaquias, M. Asavoae, and F. Brandner, "A formal framework to design and prove trustworthy memory controllers," *Real Time Systems*, vol. 59, no. 4, 2023.

[17] B. Binder, M. Asavoae, F. Brandner, B. Ben Hedia, and M. Jan, "Formal modeling and verification for amplification timing anomalies in the superscalar tricore architecture," *Int. J. Softw. Tools Technol. Transf.*, vol. 24, no. 3, pp. 415–440, 2022.