Formal Semantics of Predictable Pipelines: a Comparative Study

Mathieu Jan and Mihail Asavoae
CEA, List,
Email: Firstname.Lastname@cea.fr

Martin Schoeberl
Technical University of Denmark,
Email: masca@dtu.dk

Edward A. Lee
University of California at Berkeley,
Email: eal@berkeley.edu

***Abstract*—Computer architectures used in safety-critical domains are subjected to worst-case execution time analysis. The presence of performance-driven microarchitectures may trigger undesired timing phenomena, called timing anomalies, and complicate the timing analysis. This paper investigates pipelines specifically designed to simplify the worst-case execution time analysis (also called predictable pipelines). We propose formal and executable models of four research-oriented pipelines and one industrial pipeline to validate some of their claims related to their timing behavior. We indeed validate, via bounded model checking, the absence of a type of timing anomalies called amplification timing anomalies, or its potential presence by identifying prerequisite to situations where they can occur.**

I. Introduction

Real-time systems need to satisfy timing constraints, thus reasoning about such systems means reasoning about their timing behaviors. Time is monotonic and correct timing analyses should be constructed around this property of monotonicity. One prominent form of timing analysis, called worst-case execution time (WCET) analysis [29], computes program execution bounds and considers both the program and the underlying computer architecture. The latter is, in general, a design with performance-driven features such as pipelines, caches, and speculation mechanisms. All these microarchitecture elements may impact time monotonicity and affect, in this way, the soundness and/or the precision of a WCET timing analysis. The alternative solution to analyzing performance-driven architectures is to construct predictable ones (i.e., having a property called timing predictability [27]) and to characterize their global timing behavior by composing local timing contributions of the sub-components (i.e., a property called timing compositionality [10]).

Pipelines improve architecture performance by allowing multiple instructions to be processed at the same time as the execution of an instruction is decomposed into several computational steps. In each processor cycle, the ideal pipeline behaves as follows: an instruction completes its execution (i.e., exits the pipeline) while other instructions move one step in the pipeline and a new instruction begins its execution (i.e., enters the pipeline). The ideal pipeline preserves timing monotonicity, ensuring sequential execution of instructions. In reality, even simple pipelines can be problematic for timing analysis because of non-monotonic timing effects, also called timing anomalies [21]. Briefly, a timing anomaly is when a local worst-case timing does not lead to a global worst-case timing. For example, a restrictive pipeline, such as a classical 5-stage in-order design [13], presents timing anomalies when a first-come first-serve bus arbiter is used [10]. Predictable architectures require predictable pipelines to anchor the arguments about the system-level timing monotonicity.

In [9], the predictable pipeline of SIC has been formally proved to be without timing anomalies, while the pipelines of the predictable platforms Patmos [25] and PRET [19] are only partially formalized (the most recent PRET architecture, FlexPRET [23], is more complicated because it admits unpredictable timing behavior but for soft real-time threads). However, and to the best of our knowledge, none of the existing formal arguments of these predictable pipelines is executable. Besides, no formalization or proof of the announced predictable behavior of the K1 pipeline has been shown [5]. We advocate for formal and executable pipeline models since this synergy should be of particular interest when answering timing behavior questions. While the arguments for formalization, even on paper, are well-known, the executability aspect is often overlooked. By executable, we mean the use of model checking to animate the formalizations and proving properties about them. So far, one formal description of a predictable pipeline has been published, SIC [9], but only manually demonstrated. Besides, the executability gives way to engineering the pipeline model, in particular towards tools construction (e.g., simulators, analyzers, etc.).

Contributions. This paper makes the following contributions: a *canonical pipeline model*, to capture local worst-case timing behavior in pipeline designs; *formal models of predictable pipeline cores*, as instances of the canonical model and finally a *comparative study* of the timing compositionality of these formal pipelines. The canonical pipeline model targets amplification timing anomalies which are a requirement for sound compositional timing analyses (e.g., composing timing analyses of pipelines and shared memories). The formal models of the pipelines of PRET, Patmos, SIC, and K1, as well as of a classical in-order pipeline are based on an explicit manipulation of their stalling semantics, using the UCLID5 formal framework [26]. For SIC, we encode the formalization from [9], while for PRET, Patmos and K1, we propose what we believe to be their first formal and executable models. The comparative study assesses, on the one hand, the timing compositionality of PRET, Patmos, and SIC and on another hand, leads to the verification that only a specific configuration of K1 can be without amplification timing anomalies. Our formal models of these pipelines are available on-line on a GitHub repository [15].

The paper is organized as follows: we introduce timing anomalies, our canonical pipeline model, which focuses on the stalling logic, and its verification strategy in Section II.

978-1-7281-4123-7/20/$31.00 © 2020 IEEE

In Section III, we then present our formal models of the predictable pipelines SIC, PRET, Patmos, as well as the K1. We describe the results of their evaluation, via model checking, in Section IV. We address related works in Section V. We conclude and outline future works in Section VI.

II. Formalizing Stalling of Pipelines Design

In this section, we briefly introduce timing anomalies and our abstract canonical pipeline model to facilitate the comparison between pipelines (e.g., what are the necessary assumptions to represent the memory system or the ISA-level restrictions etc.). We then present our verification strategy.

A. Timing Anomalies

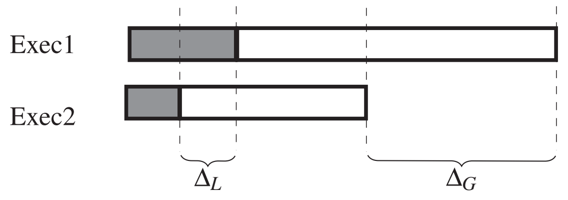

Fig. 1. Example of an amplification timing anomaly.

A timing anomaly defines (1) a counterintuitive timing behavior, or (2) a timing amplification. The counterintuitive timing anomaly appears when a reduced local timing leads to an increased global timing [21]. A local timing variation refers to the possible latencies of an instruction, due to memory access variability, for instance. The global timing variation refers to the possible execution times of a sequence of instructions. The presence of such counterintuitive timing anomalies leads to *timing predictability* problems and requires an exhaustive exploration of all possible behaviors to estimate a safe WCET bound. It can occur whenever a run-time decision has to be made for selecting the execution unit used to execute an instruction [28]. In-order pipelines could exhibit such behaviors if an instruction requires multi-cycle latencies in its execution stage and a run-time decision for choosing a functional unit. These timing anomalies have been investigated in [1], while in this paper we focus on (2).

The second timing anomaly, the amplification timing anomaly, is shown in Fig. 1. It appears when a local timing variation of Δ_L leads to a larger global variation of Δ_G. The presence of amplification timing anomalies threatens the *timing compositionality* (a property formally defined in [10]) which is necessary to address the timing analysis of multi-component systems. For example, timing compositionality is required to combine the cache-level timing analyses with pipeline-level ones when performing a WCET analysis. In-order pipelines still suffer from amplification timing anomalies when they use a first-come-first-serve bus arbiter, as shown in [10]. The local timing variation of the described amplification timing anomalies is due to a switch from a cache hit to a cache miss for a first instruction. This local variation then triggers an increased global variation, the bus being occupied by another instruction (succeeding the first instruction in the execution flow of a sequence of instructions) stalling the cache miss request of the first instruction.

The absence of amplification timing anomalies is a necessary and sufficient condition for achieving timing compositionality. When timing compositionality is not ensured, a WCET analysis can not be safely decomposed per component and thus requires over-approximations. The soundness of a WCET analysis then requires bounds on the potential amplification timing anomalies in the system. However, note that if an amplification timing anomaly occurs at each iteration of a loop, it cannot be bounded by a constant value but by a value proportional to the loop bound. Our current approach aims towards the detection (and bounding) of such amplification timing anomalies.

B. Canonical Pipeline Model

The goal of our canonical pipeline model is to focus on amplification timing anomalies linked to bus accesses to shared memory. For each pipeline, we define a set of instructions classes and a set of pipeline stages. All instructions of an instruction class must have the same temporal behavior over all the pipeline stages. The instruction classes that we define are: *load*, *store*, *branch*, *nop*, and *other*. *Branch* and the optional *nop* instruction classes enable the modeling of instructions of these classes becoming inactive after a given stage. Pipeline stages are specific to each pipeline. However, we introduce two additional stages *pre* and *post* to model an instruction, respectively, waiting to enter the pipeline, and that has finished its execution.

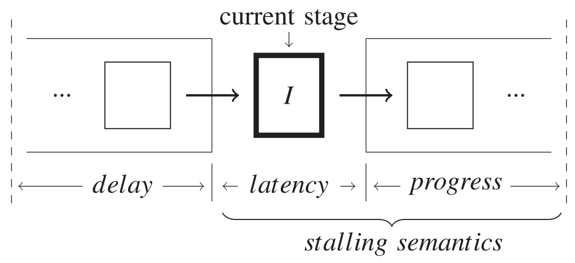

Fig. 2. Canonical pipeline model: the advancement of instruction *I* in the pipeline depends on *delay*, *latency*, and *progress*.

Our canonical model instantiates two instructions, called *upstream* and *downstream* in the pipeline. The *downstream* instruction *precedes* the *upstream* instruction in the execution flow of a sequence of instructions. The downstream instruction thus enters first in a pipeline, thus reaching pipeline stages before the upstream instruction.

An instruction state is defined as:

$$\langle type,\ vl,\ stage,\ latency,\ delay,\ progress \rangle$$

where *type* is the instruction class, *vl* is a vector with initial latencies in each stage, *stage* is the current stage, *latency* is its remaining latency in *stage*, *delay* is the accumulated delay (due to stalling) up to *stage*, and finally, *progress* is a boolean which captures the decision on the stalling logic. Each of these variables is subscripted with *down* or *up* to identify respectively downstream and upstream instruction states. Note that all timings are expressed in clock cycles and that our canonical model is cycle-accurate. Data dependencies between instructions are not modeled, as they do not generate timing anomalies.

For each instruction, two procedures are defined. The procedure *next_stage* computes the next stage of an instruction and updates its $latency$, with the one initially specified for that next stage (noted $vl.stage$). The procedure *check_progress* computes the decision of the stalling logic being modeled, only if $latency$ has elapsed. In the next section, we present our baseline stalling logic, while Section III details the stalling logic of each pipeline we have modeled.

C. Baseline Stalling Logic

Our canonical pipeline model relies on an encoding of the stalling semantics using the next operator from UCLID5. The baseline stalling logic is implemented as in a textbook in-order processor pipeline [12]. This pipeline is organized into five pipeline stages: (1) instruction fetch (IF), (2) decode and register read (ID), (3) execute (EX), (4) memory access (MEM), and (5) register write back (WB). We assume separated instruction and data caches, and thus two potential cache misses and stalling from IF and MEM stages. In these cases, the next stage of the downstream instruction is by design available in our model. However, the next stage of the upstream instruction might be occupied by the downstream instruction. The *check_progress* procedure thus takes as input $stage'_{down}$ ($'$ denotes the next state of a variable) to stall the upstream instruction in that case. For both instructions, the *check_progress* procedure also checks whether the bus is already used, due to a miss, by the other instruction. In that case, the $delay'$ of the instruction being stalled is incremented. The upstream instruction can progress whenever its next targeted stage is available when $stage_{up} \neq pre$ or the following condition holds:

$$stage_{up} = pre \Rightarrow stage'_{down} \notin \{IF\} \wedge \\ (stage'_{down} \notin \{MEM\} \vee vl.MEM_{down} = 1)$$

i.e., the downstream instruction is neither at the IF stage, nor it generates a data miss from the MEM stage. Finally, note that we model a higher priority for data accesses over instruction accesses, as commonly implemented within real pipelines. This stalling logic is called *specific upstream*, as the upstream instruction is stalled, due to memory access, only when it is at a specific stage, in our case at the pre stage.

We have also implemented another stalling logic, called *only upstream*, which stalls the upstream instructions at their current stages, whatever they are, when a cache miss occurs. Downstream instructions can progress meanwhile. The previous specific upstream logic is checked at any stage of the upstream instruction.

Another stalling logic stalls the whole pipeline thus called *whole*. Preceding instructions, within the sequence of instructions, the one causing the stall are then also stalled. This whole pipeline stalling logic is implemented by Patmos, and thus we detail it later.

D. Verification Strategy

Our verification strategy for our canonical model explores the localization of all upstream instructions in the pipeline that can impact/stall the downstream instruction. To achieve this, we systematically explore, using the bounded model checking of UCLID5, the following: (1) all possible instruction classes combinations of upstream and downstream instructions, (2) all possible distances between upstream and downstream instructions and, finally, (3) all possible memory latencies, within a bounded range, for the upstream and the downstream instructions. Precisely, point (1) is trivially setup by assuming that each instruction can be of any class; for point (2), the downstream instruction can start from any possible pipeline stage; and finally, for the point (3), the latency of memory accesses varies between 1, a cache hit, and 10 cycles, a cache miss (i.e., a standard ratio for the cache hit-miss). Interested readers may refer to [3] for an introduction to bounded model checking.

A range of latencies is indeed needed to explore all the scenarios where the upstream instruction enters the pipeline while the downstream instruction is waiting for a memory transfer to finish. This memory transfer could be generated by the downstream instruction, but also by the upstream instruction from the IF stage. The chosen hit/miss latencies are classical values for the embedded systems we target that have a limited memory hierarchy, i.e., a private L1 and a shared L2. For instance, the commercial K1 core has a cache miss penalty, from a private L1 cache towards a shared L2 SRAM, of around 10 clock cycles. The Patmos, configured for the default FPGA boards, has a memory latency towards an SRAM of 21 clock cycles. Finally, note that path-level semantic, due to branches, has no impact on our canonical model as our verification strategy explores all combinations and distances of upstream and downstream instructions.

III. Formal Predictable Pipelines

SIC enhances the compositionality of the classical 5-stage in-order pipeline with specific rules to enforce the timing monotonicity w.r.t. the memory system [9]. The memory accesses of both the IF and MEM stages are indeed constrained to be performed in the program order. The in-order stalling logic is thus extended with an additional condition preventing the upstream instruction to enter the pipeline while a memory operation is pending. For loads, it is as follows:

$$stage_{up} = pre \Rightarrow \neg mem\_pending(stage'_{down})$$

with $mem\_pending = (type_{down} = load \wedge vl_{down}.MEM > 1 \wedge stage'_{down} \in \{IF, ID, EX, MEM\})$. A similar condition holds for stores. The downstream instruction is no longer stalled in the MEM stage due to a pending instruction cache miss generated by the upstream instruction from the IF stage. Dependencies between instructions leading to amplification timing anomalies are thus removed. Besides, we introduce a ST stage in the *next_stage* procedure to model the asynchronous semantic of stores assumed in [9].

PRET, the Precision Time architecture [20] targets the specification of multithreaded processors able to execute several hard real-time tasks concurrently. It relies on a thread-interleaved in-order five-stage pipeline to exploit the thread-level parallelism. A static round-robin scheduling is used to fetch, at every cycle, an instruction from a different hardware thread (amongst 4) enabling a constant latency between successive instruction fetches for the same thread. All data and instructions are assumed to be initially stored in a scratchpad memory, whose access latency is 1 cycle. Modeling PRET is thus straightforward as there is simply no need for a stalling logic. Compared to the classical in-order pipeline, the *progress* condition is thus always true, i.e., there is no need to check whether the next stage is available.

Patmos [25] is a five-stage, in-order pipeline that can be connected to a method cache [6], instead of a regular instruction cache. Using this method cache, misses may only occur in the *MEM* stage when loading instructions on a call or return: an instruction fetch is guaranteed to always hit. We thus include instructions handling the method cache in the *load* instruction class. Stalling due to method cache but also data cache misses can only occur from a single stage within the pipeline: *MEM*. This simplifies the hardware design of the *whole* stalling logic. Compared to the *only upstream* logic, the downstream instruction now stalls when the following condition holds:

$$(stage_{up} = MEM \land vl.MEM_{up} > 1 \land latency_{up} \geq 1)$$

As misses occur only in the *MEM* stage, the downstream instruction can only be stalled at the *WB* stage. This stalling logic enforces a fully timing-compositional behavior.

K1 core is a 7-stage pipeline stage where a sequence of four execution stages, named $E1$ to $E4$, take place after Instruction Decode (*ID*) and Register Read (*RR*) stages. A prefetch buffer connects the pipeline to an instruction cache, that we modeled as a regular *IF* stage. We divide the *other* instruction class into *mac* (multiply and accumulate operations) and *alu* (arithmetic and logical operations). The *alu* instruction class spends only 1 cycle at the $E1$ stage, while the *mac* stays 1 cycle in each of the four Ex stages. In case a miss occurs, loads stall at the $E3$ stage. Stores are asynchronous, with thus a latency of 1 cycle in their last $E3$ stage.

K1 implements a variant of the *specific upstream* stalling logic. The upstream instruction is stalled at the $E1$ stage. However, this stalling occurs only for loads or stores. Besides, memory transfers by the downstream can span over several stages, leading to the following stalling condition for the upstream instruction:

$$\begin{aligned}&type_{up} \in \{load, store\} \land \\ &\quad ((stage'_{down} \in \{E1, E2, E3\} \land (type_{down} = store \\ &\qquad \lor (type_{down} = load \land vl.mem_{down} > 1))) \\ &\qquad\qquad\qquad\qquad \lor unv\_dcache > 1)\end{aligned}$$

Checking if the downstream instruction is a store is required, as we assume the default write-through policy for the data cache. Finally, the last variable (*unv_dcache*) models the cache being unavailable for some extra-cycles: (1) to evict data from the cache and (2) to implement the critical-word-first return strategy used to load the core with the initially missed-data. The advantage of this strategy is that instructions can be released as soon as possible. For (1), we arbitrary select a value, as the number of extra clock cycles depends on the availability of the shared memory. For (2), we add 3 more extra clock cycles to any load miss, to align on the cache line size. The variable *unv_dcache* is updated after the next state of the downstream instruction is computed. It can thus impact the stalling logic of the upstream.

Both Patmos and K1 cores are n-issues Very Long Instruction Word (VLIW) pipelines. For both of them, only a single pipeline can generate memory accesses, allowing us to ignore modeling bundles of instructions.

IV. Properties and Results

Each pipeline model has been verified using specific tests and the verification strategy presented in II-D. These runs were performed with 2 invariants: (a) $stage_{down} \neq stage_{up}$, except when they are in the *pre* and *post* stages and (b) the latency for both instructions must always be valid, i.e. $latency_{down} \geq 0 \land latency_{up} \geq 0$. The depth of the Bounded Model-Checking (BMC) is set to the minimal value that ensures that both the upstream and downstream instructions have been fully executed, i.e. have reached the *post* stage. To this end, the BMC depth is first manually set to an arbitrary value. Then, we manually iterated until we found this minimal value ensuring that the following Linear Temporal Logic (LTL) property is verified:

$$\mathbf{G}(step = depth \Rightarrow (stage_{down} = post \land stage_{up} = post)) \quad (1)$$

where *step* is the current step and *depth* is the BMC bound. The *Globally* temporal operator **G** of LTL applied to a property p, i.e. $\mathbf{G}p$, means that p holds in all states. Performing this verification, following our strategy presented in Section II-D, can take from a few minutes, for PRET, to several hours, for K1, and has shown to detect bugs in the early design phase of these models.

After these first set of runs, we then focus on checking whether a downstream instruction can or cannot be stalled by an upstream instruction. This is achieved by computing in our models the delay of the downstream instruction ($delay_{down}$), i.e., its stalling time due to upstream instruction. Whether the upstream instruction can delay the downstream instruction is then simply verified using the following LTL property:

$$\mathbf{G}(delay_{down} = 0) \quad (2)$$

This property is a prerequisite to situations where amplification timing anomalies might occur and thus not amplifications directly. This LTL property is thus only a sufficient condition, and false positives can be found, i.e. non-null delays computed but no amplification timing anomalies, as shown later, when we report the results. Directly checking amplification timing anomalies would require to compare local variations of the latency of instructions to the global variation of the execution time of a sequence of instructions. Note that for these second set of runs, we use the same strategy as in the first set of runs to set the depth of the BMC, i.e., the LTL property 1 is combined with property 2. Finally, note that the BMC depth is not directly related to the complexity of a WCET analysis. However, the absence of timing anomalies, i.e., both amplification and counterintuitive, allows to simply aggregate local timing contributions into a WCET analysis, i.e., an exhaustive search is no longer required and thus the complexity of a WCET analysis is reduced.

Table I shows, for each pipeline, whether prerequisite to amplification timing anomalies are identified or their absence proved (noted *No*), the required BMC bound and the runtime of the verification process. These runtimes are the total times

TABLE I
IDENTIFIED OR PROVED ABSENCE OF PREREQUISITE TO AMPLIFICATION TIMING ANOMALIES FOR PIPELINES.

Pipeline (stalling logic)	Prerequisite to amplifications	BMC *depth*	Runtime (s)
In-order (specific/only)	Identified	31	44.5
In-order (whole)	Identified	34	83.8
SIC (specific)	No	23	41.1
PRET (no)	No	10	9.4
Patmos (whole)	Identified	14	12.2
Patmos (specific)	No	14	11.5
K1 (∼specific)	Identified	33	84.8

to verify the properties 1 and 2 over each model and not of WCET analyses of given programs over our pipeline models. These runtimes cannot be directly compared to any previous work, as none has targeted the verification of timing amplifications anomalies. Verification runtimes reported in [16] are only a few seconds, but indeed ignore to model the interplay between instructions, as we do in our canonical model. Compared to [17], our runtimes are this time much lower as we focus on a specific timing behavior and not on a set of functional properties.

As expected, the simplest pipeline in which the absence of amplification is proven is PRET, with BMC *depth* of 10. Patmos requires a BMC depth of 14. However, a set of situations with a positive *delay* are identified. These cases correspond to the situations where the downstream instruction is stalled at the *WB* stage, due to a cache miss generated by the upstream instruction at the *MEM* stage. We have checked that these counter-examples, due to the whole stalling logic, cannot lead to amplification timing anomalies. The downstream instruction does not indeed experience a local timing variation in its latency at the *WB* stage. The delay the downstream instruction is stalled in these cases, and thus the global timing, must include the worst-case latency of the upstream instruction at the *MEM* stage. Additionally, we have verified that a Patmos model using the *specific upstream* stalling logic is proved to be without timing amplifications. The additional specific rules of SIC requires an increased BMC depth of 23. This is due to other rules in the model to ensure compositionality. We have omitted them in our description of Section III, due to space constraints.

Finally, a prerequisite to amplification timing anomalies are identified for both in-order (independently of the stalling logic) and K1 pipelines. For both pipelines, they correspond to a local variation due to a switch from a cache hit to a cache miss at the *MEM* stage of the downstream instruction. The increased global variation then comes from the bus being occupied by the upstream instruction at the *IF* stage, stalling the cache miss request from the downstream instruction. K1 requires a higher BMC depth due to the specific behavior of its data cache. We also modeled the streaming mode of the K1, where instructions can progress up to a fifth pending uncached load reaches the $E3$ stage. The uncached streaming stores are supported in an unlimited number. Under these conditions, we verify that K1 does not exhibit timing amplifications.

To summarize, we establish the following complexity-based ordering between the proven, predictable pipelines: $PRET < Patmos < SIC$. SIC has a complexity similar to a classical in-order pipeline, while Patmos almost reaches the simplicity of PRET but brings additional specialized caches into play.

All the developed models are in open source and available on-line on a GitHub repository [15]. The `README.md` provides instructions on how to rerun our experiments.

V. RELATED WORK

WCET analyses have been the subject of numerous publications, [29] provides an overview of the subject to interested readers. However, the notion of timing anomaly, and implicitly its first (semi-formal) definition, is introduced in the context of the WCET analysis in [21]. It is further refined (and accompanied by a simple detection criterion), in [28] and finally, it is formally defined in [24]. Several existing WCET tools still assume no timing anomalies, such as Heptane [11].

Several approaches are proposed towards formally reasoning about counterintuitive timing anomalies in pipelined systems. For example, in [7], the absence of such anomalies is verified, using bounded model checking and in [1], guided model-checking is combined with a state-space transformation to detect such anomalies. The approach in [8] augments a static analysis with measurements executed directly on the system under analysis. In this work, we use the bounded model checking techniques of UCLID5 [26] to focus on amplification timing anomalies.

The formal modeling and verification of computer architectures in general, and pipelines in particular, has been addressed in numerous works. A wide range of solutions have been proposed, from manual to automated/synthesized correctness proofs, addressing in general the functionality correctness. Without being exhaustive, we relate to several approaches which, roughly, fall into three categories: model checking and decision procedures [4], [17], [22], theorem proving [16], [18] and mixed techniques [14]. The work in [4] couples a stalling semantics with a flushing semantics (which is also based on pipeline stalls) to prove that instructions are correctly executed (i.e., functional correctness). [16] focuses on the stalling semantics of simple in-order processor (as ours) coming from data hazards or a slow (data) memory, ignoring the instruction memory. [2] extends the scope of this work to a whole architecture but still without considering the interplay between instructions and data memories on the stalling semantics of pipelines, as we consider in our work. The same (limited) stalling semantics of in-order pipeline is synthesized in [17], whereas, the stalling semantics from [22], [18] and [14] are for out-of-order pipelines, thus richer than what we propose. However, the particularity of our stalling semantics is the canonical pipeline model – designed to track local timing variations and hence, for worst-case timing analysis.

VI. CONCLUSIONS AND FUTURE WORK

In this work, we proposed a formal and executable framework to facilitate the automated detection of amplification timing anomalies. We used UCLID5 to encode a classical in-order pipeline and several predictable pipelines. We validated, via bounded model checking, the absence of such anomalies for the PRET, Patmos and SIC pipelines. Finally, we have shown that K1 is subjected to these timing anomalies, except if streaming accesses are used.

We are currently working on how to generate such models from hardware description languages and how to define appropriate abstractions over these models. Besides, it could be interesting to extend our canonical pipeline model to generate the scheduling parameters of hard and soft real-time threads for FlexPRET. We also plan to extend our canonical model, by instantiating several instructions, to verify both amplifications and counterintuitive timing anomalies.

References

[1] M. Asavoae, B. B. Hedia, and M. Jan. Formal executable models for automatic detection of timing anomalies. In *18th Intl. Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 2:1–2:13, Barcelona, Spain, 2018.

[2] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together – formal verification of the vamp. *International Journal on Software Tools for Technology Transfer*, 8(4):411–430, Aug 2006.

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification, 6th International Conference, CAV '94*, pages 68–80, 1994.

[5] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Proc. of the Conf. on Design, Automation & Test in Europe (DATE'14)*, pages 97:1–97:6, 2014.

[6] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl. A method cache for Patmos. In *Proc. 17th Intl. Symp. on Object/Component-Oriented Real-Time Distributed Computing*, ISORC '14, pages 100–108, Washington, DC, USA, 2014.

[7] J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proc. of the Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 15–20, Prague, Czech Republic, April 2006.

[8] G. Gebhard. *Static timing analysis tool validation in the presence of timing anomalies*. PhD thesis, Saarland University, 2013.

[9] S. Hahn and J. Reineke. Design and analysis of SIC: A provably timing-predictable pipelined processor core. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 469–481, Nashville, TN, 2018.

[10] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015.

[11] D. Hardy, B. Rouxel, and I. Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 8:1–8:12, 2017.

[12] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[14] C. Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *14th Intl. Conf. Computer Aided Verification (CAV)*, pages 309–323, 2002.

[15] M. Jan. UCLID5 models of SIC, PRET, Patmos and K1, 2019. https://github.com/t-crest/patmos-sail/tree/master/uclid.

[16] D. Kröning. *Formal verification of pipelined microprocessors*. PhD thesis, Saarland University, Saarbrücken, Germany, 2001.

[17] U. Kühne, S. Beyer, J. Bormann, and J. Barstow. Automated formal verification of processors based on architectural models. In *10th Intl. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 129–136, 2010.

[18] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002*, pages 142–159, 2002.

[19] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *30th Intl. Conf. on Computer Design, ICCD*, pages 87–93, 2012.

[20] I. Liu, J. Reineke, and E. A. Lee. A PRET architecture supporting concurrent programs with composable timing properties. In *44th Asilomar Conf. on Signals, Systems, and Computers*, November 2010.

[21] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proc. of the 20th Real-Time Systems Symposium*, pages 12–21, Phoenix, AZ, USA, December 1999.

[22] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *10th Intl. Conf. Computer Aided Verification (CAV'98)*, pages 110–121, 1998.

[23] C. S. Michael Zimmer, David Broman and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Berlin, Germany, April 2014.

[24] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dresden, Germany, July 2006.

[25] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[26] S. A. Seshia and P. Subramanyan. Uclid5: Integrating modeling, verification, synthesis and learning. In *16th Intl. Conf. on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–10, Oct 2018.

[27] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[28] I. Wenzel, R. Kirner, P. P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Intl. Conf. on Quality Software (QSIC 2005)*, pages 295–306, Melbourne, Australia, September 2005.

[29] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.