Is This Still Normal? Putting Definitions of Timing Anomalies to the Test

Benjamin Binder*, Mihail Asavoae*, Belgacem Ben Hedia*, Florian Brandner† and Mathieu Jan*
**Universit Paris-Saclay, CEA, List*
†*Institut Polytechnique de Paris, Tlcom Paris, LTCI*
F-91120, Palaiseau, France

***Abstract*—Correctness is an important concern during the development of real-time systems. In addition to the functional correctness, the timing behavior is often formally verified in order to ensure that correct results are delivered in-time for all possible execution conditions. The timing behavior of real-time software is thus often validated through a rigorous *timing analysis* that aims at determining the *worst-case execution time*.**

Timing anomalies present a major obstacle during the validation of timing properties on modern computer platforms. Out-of-order execution and concurrent accesses to shared resources may sometimes lead to – at first sight – *surprising* timing behavior. Several (semi-)formal definitions have been proposed in the literature in order to capture such situations. However, as we present in this work, none of the existing definitions appears to be precise enough to be systematically used for detecting timing anomalies in modern processors with out-of-order execution.

***Index Terms*—Formal Methods, Model Checking, Timing Anomalies**

I. Introduction

Hard real-time systems are often subject to certification. This means that the correctness of the system has to be verified in terms of the functional correctness as well as the system's timing behavior. The timing behavior can be validated through static analysis [1], test-based measurements [2], or probabilistic analysis [3], [4], which all try to determine a *tight* estimation of the *Worst-Case Execution Time* (WCET) of a piece of real-time software under all possible execution conditions (in the latter case possibly assuming a given target probability). None of these approaches is able to explore all possible executions and thus only provide safe WCET estimates when certain underlying hypotheses are satisfied.

Timing Anomalies (TA) pose a challenge to all three approaches due to their impact on these hypotheses. Intuitively, a TA is a (local) condition at a given moment during the execution of real-time software that leads to a *surprising* effect on the (global) execution time. A *counter-intuitive* TA occurs when the local condition is favorable (respectively unfavorable) in terms of timing, e.g., a cache hit (miss), but leads to an increase (decrease) of the global execution time. *Amplification* TAs are situations where a local slowdown (respectively speedup), e.g., a cache miss (hit), leads to a proportionally larger increase (decrease) of the global execution time. We generally retain from the above descriptions of TAs the sense where the execution time is increased, since these TAs may have an impact on the WCET of real-time software.

TAs are problematic for static analyses because it is no longer safe to consider only the local worst-case conditions. Instead, an *exhaustive exploration* of the reachable hardware states has to be performed, which is costly or often even prohibitive. The same issue arises for test-based approaches relying on measurements, since the possible number of tests to cover increases drastically. Probabilistic methods do not fare better. Slight changes in the hardware state may trigger a TA, which in turn may cause a considerable increase of the execution time. TAs may thus invalidate fundamental hypotheses of probabilistic approaches (e.g., independence and stationarity) and thus pose a threat to the validity of the obtained results [4].

TAs represent a major challenge for the real-time systems community and consequently have gained quite some attention in recent years. However, the usual interpretation of the term *Timing Anomaly* remains rather colloquial and the understanding of the underlying effects is often only illustrated through examples that provide some *intuitive* understanding. Existing work is often incomplete, providing abstract notions and making it difficult to implement the definitions – let alone reason about TAs – on concrete software and/or hardware. Moreover, most definitions are based on generic transition systems and thus do not restrict in any way the kind of processor that is modeled. Notably, the conditions under which these definitions apply are not restricted. Hence, any procedure for detecting TAs directly derived from them should be exact, excluding false positives/negatives.

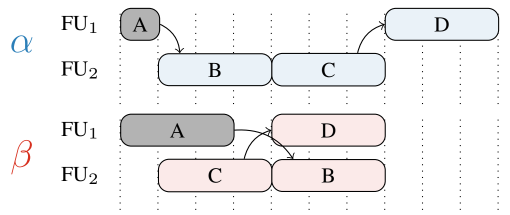
The goal of this work is thus to **put existing definitions to the test**. Are those definitions able to capture the intuitive understanding of TAs established in the community? Do those definitions provide reliable and coherent answers when applied to different execution scenarios? In order to answer those questions, we have: 1) **developed a parameterizable *formal* model of a standard out-of-order (OoO) processor**; 2) **encoded the most relevant formal definitions of TAs into *executable* procedures coupled to the processor model**; 3) **then assessed the definitions through model checking by finding examples that lead to contradictions** (among those definitions). Our assessment[1] shows that *no definition* is able to identify TAs precisely on all the considered examples.

The rest of this paper is organized as follows. In Section II,

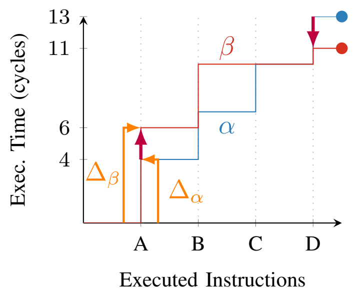
[1]Sources are available at: https://bitbucket.org/benjaminbinder/ta-models/

```
⤹ LD    r1, 0(r2)    ; A
  ADD   r3, r1, r4   ; B
⤹ ADD   r5, r6, r7   ; C
  LD    r8, (0)r5    ; D
```

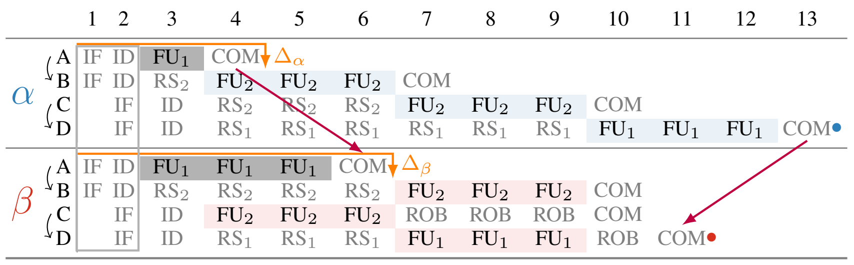
(a) Input assembly code



(b) Scheduling on functional units (FUs)



(c) Execution time as step function



		1	2	3	4	5	6	7	8	9	10	11	12	13
α	A	IF	ID	FU$_1$	COM Δ_α									
	B	IF	ID	RS$_2$	FU$_2$	FU$_2$	FU$_2$	COM						
	C		IF	ID	RS$_2$	RS$_2$	RS$_2$	FU$_2$	FU$_2$	FU$_2$	COM			
	D		IF	ID	RS$_1$	RS$_1$	RS$_1$	RS$_1$	RS$_1$	RS$_1$	FU$_1$	FU$_1$	FU$_1$	COM●
β	A	IF	ID	FU$_1$	FU$_1$	FU$_1$	COM Δ_β							
	B	IF	ID	RS$_2$	RS$_2$	RS$_2$	RS$_2$	FU$_2$	FU$_2$	FU$_2$	COM			
	C		IF	ID	FU$_2$	FU$_2$	FU$_2$	ROB	ROB	ROB	COM			
	D		IF	ID	RS$_1$	RS$_1$	RS$_1$	FU$_1$	FU$_1$	FU$_1$	ROB	COM●		

(d) Execution traces showing latencies (⤵), the order of commits (→), the assignment to functional units (🟦/🟥/⬜), and the end of traces (●/●).

Fig. 1: Different ways of representing two execution traces that constitute a counter-intuitive TA (1b, 1c, 1d) on an out-of-order (OoO) processor from a given program (e.g., 1a) showing data dependencies (⤹).

we introduce various definitions of TAs found in the literature, as well as the necessary assumptions to interpret them. In Section III, we present our model for tracking TAs in OoO processors, under our implementations of these definitions, and how we use model checking to assess them. Then, we probe the definitions in Section IV, through examples that highlight their short-comings. In Section V, we discuss additional related work. Finally, we discuss our assessment of the definitions and outline remaining work in Section VI.

II. Background and Interpretations

Lundqvist and Stenström first introduced the notion of TAs [5], from the observation of different behaviors of a processor when executing the same program, namely (execution) *traces*. Their definition is based on instruction sequences whose first instruction has a variable latency, e.g., due to a cache hit/miss. They define the notion of *anomalies* by comparing two execution traces and provide examples for an OoO processor. The definition is incomplete as it only allows for a *single* latency variation at the first instruction and does not define instruction latencies. Wenzel et al. adopt the same framework [6]. Though latencies are clearly defined as the time spent in functional units, their definition is still restricted to a single variation. Moreover, the definition demands "almost identical" initial hardware states, without a clear definition.

In the following we consider a list of more recent definitions for which we were able to develop formal models. In order to make them applicable and evaluate their capabilities in the detection of TAs, we need to rely on a set of interpretations (which are to be detailed and explained when needed). We will focus on the formal definitions of *counter-intuitive* TAs, which are dominant in the literature. For each of them, first, we introduce the definition before illustrating it through an example. Then, we highlight problems that we encountered while trying to encode the definition in a *systematic* manner and we establish assumptions that enabled us to encode it.

The example of this section is presented in Fig. 1. It is based on a commonly used representation (Fig. 1b) of a counter-intuitive TA caused by a variation in the number of cycles that instruction A spends in functional unit FU$_1$, which impacts the instruction scheduler of an OoO processor executing a sequence of instructions such as the one of Fig. 1a with (read-after-write) data dependencies. In Fig. 1d, we detail the two execution traces *assuming a concrete processor model*, i.e., a dual-issue processor with OoO execution on two functional units (FUs) associated with reservation stations (RS) and in-order commit (COM) through a reorder buffer (ROB).

A. Step Height in Step Functions

A simple definition of TAs is provided by Gebhard [7]. The evolution of traces (cumulative execution time) can be represented as a *step function* (Fig. 1c). According to Gebhard, a TA could thus be identified by comparing the step functions of two traces: a TA occurs when the latency for an instruction in one trace, i.e., the corresponding step height in the plot, is smaller than in the other trace, but later the execution time is larger.

This is illustrated by Fig. 1c, where the step height Δ_α of trace α is smaller than Δ_β of trace β, but the execution time of α at the end of the trace is larger than that of β (●$13 > 11$●).

However, the definition leaves several details open. For instance, it relies on the notion of hardware states without a clear definition. The same applies to instruction latencies, which are only supposed to be non-negative and yield the execution time when summed. Instruction latencies are obvious on in-order processors, but the situation is more complex for OoO processors. The notion of latencies used by Wenzel et al. [6], for instance, is not admissible due to the second constraint.

For this approach, we thus assume instruction latencies as the number of cycles between the commit of an instruction (COM) and the previous commit (or trace start) according to the program order of instructions. The instruction latencies for trace α in Fig. 1d are thus 4, 3, 3, and 3 cycles for instructions A through D respectively. This matches the step function in Fig. 1c. Finally, note that we choose not to include borderline cases with equal global times in both traces as TAs, since it is not a strict inversion and it does not impact the WCET.

B. Intersections in Step Functions

An alternative definition of TAs, which also relies on the notion of step-functions, was proposed by Kirner et al. [8] (later Cassez et al. [9]). A TA occurs when the step functions of two traces *intersect*, i.e., a trace that initially executed instructions *faster* suddenly becomes *slower* than the other.

This situation is illustrated in Fig. 1c, where the traces intersect at the last instruction: α initially situated below β passes above. From the existence of such an intersection it follows that the absolute values of the step functions switch order, as indicated by the red arrows ($\uparrow$and$\downarrow$). Such an inversion can also be observed in the detailed execution traces from Fig. 1d by looking at the respective instances at which instructions were committed (COM). Instruction A, for instance, was committed in cycle 4 for α but in cycle 6 for β, as illustrated by the red diagonal arrow. The situation is inverted for instruction D, as indicated by the red arrow pointing in the opposite direction. Red arrows pointing in opposite directions ($\searrow$ vs. $\swarrow$) then indicate a TA (similarly to *intersections* in plots).

Another *equivalent* definition of TAs is used by Eisinger et al. [10]. The only difference wrt. the above is that the two axes of the step function are switched, i.e., the number of instructions completed in an arbitrary time window is tracked.

C. Component Occupation

The previous definitions summarize the timing of individual instructions using a single value, the instruction latency. Kirner et al. [11] propose a different view that focuses on the usage of a hardware component (resulting from a partitioning of the whole architecture) throughout a trace. By comparing the amount of time a FU is occupied among two traces along with the traces' execution times, they describe a new type of TAs called *parallel inversion*.

The usage of FU_1 for the traces depicted in Fig. 1d, for instance, amounts to 4 and 6 cycles for α and β respectively. Despite the higher usage of FU_1, β's execution time is lower, which indicates a TA. The usage of FU_2 is the same in both traces and thus does *not* indicate a TA. A component including both FUs again yields a TA.

The main problem with this definition is that the authors do not describe what a suitable hardware component is, e.g., whether/how FUs have to be grouped together in a single component. As illustrated by the example from above, the hardware partitioning has an impact on the identification of TAs. Unfortunately, the authors do not describe how to obtain partitions that reliably identify TAs.

We assume any non-empty subset of the FUs to be a possible component. In our formal model, the considered subset is supplied as an input parameter and we assume that a TA exists if a parallel inversion is signaled for the FU(s) of this subset.

D. Instruction Locality

Reineke et al. [12] propose yet another point of view, making it possible to mix per-instruction latencies with the notion of occupation of a FU—the so-called *locality*. It is based on a *transition system* that specifies the *cycle-level* behavior of the considered processor. The authors assume that one can derive an assignment of instructions to FUs from a given state of the transition system. Consequently, it is possible to extract on every cycle the *locations* (FUs or hardware components) occupied by an instruction. The authors also assume that the occupation of locations by an instruction may change due to *non-deterministic* behavior. TAs are then defined by comparing the occupation of locations *by an instruction* at the first instant when two execution traces diverge, along with the traces' execution times.

Consider again Fig. 1d in order to illustrate the situation. The depicted traces are clearly identical for the first two cycles as indicated by the gray box on the left (▢). In cycle 3, a non-deterministic choice causes the traces to diverge. As a result, FU_1 is occupied by 1 vs. 3 cycles for α and β respectively (■), which has an impact on the subsequent instruction scheduling and leads to different execution times. The TA is explained by the fact that instruction A occupies FU_1 longer in β, thus called a *local worst-case* trace, while yielding a shorter execution time.

A first problem with this definition concerns the local occupation of locations, which are described as *convex predicates*. The authors do not explain how to chose suitable locations (only that they should be at the pipeline-stage level) nor how to obtain corresponding convex predicates. Then, it is unclear how to compare the occupation once the states of the transition system for two traces have diverged, e.g., when an instruction occupies different locations in the two traces in the next cycle. For this work, we assume that the transition system can be represented by a table, similar to the one from Fig. 1d, that assigns instructions to pipeline stages. We assume that every instruction occupies only one location at a time in a trace (cf. convex predicates). Since the RSs and the ROB only model instructions waiting for a FU or for committing, i.e., due to the scheduling of other instructions, we furthermore assume all pipeline stages except them to be part of the set of suitable locations. We also assume that FUs are gathered in a single execution stage as a suitable location. This allows

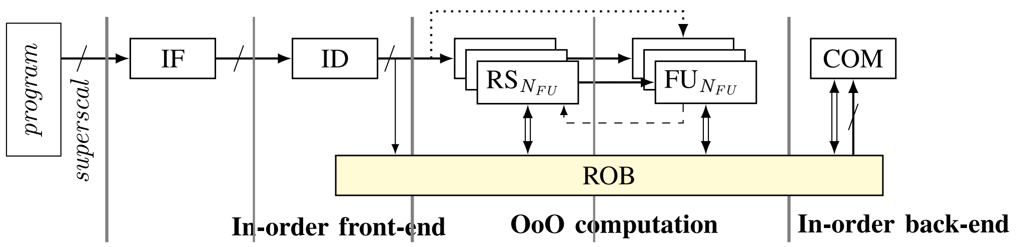


Fig. 2: Parameterizable OoO pipeline with N_{FU} functional units, fetching and committing $superscal$ instructions per cycle.

(local) comparisons of the occupation even when different FUs are used for the execution of an instruction. Finally, we do not compare the occupation of locations after the moment when the traces diverge.

III. Formal Modeling of TAs in OoO Processors

In this section, we present our *formal* and *executable* model to evaluate the presence or absence of TAs under the various definitions described above. We aim at making this assessment by model checking, over executions of specific traces on standard OoO architectures.

A. Parameterizable Out-of-Order (OoO) Pipeline Model

We base our work on a parameterizable hardware architecture template, similar to previous work [13], which is representative of modern in-order and out-of-order pipelines that are susceptible to TAs.[2] OoO execution is implemented using Tomasulo's algorithm [14] with a reorder buffer (ROB) that ensures in-order completion. The architecture template, illustrated by Fig. 2, is composed of:

- an in-order front-end in charge of fetching (IF stage in Fig. 2) and decoding (ID stage) instructions, where the maximum number of instructions that can be fetched/issued is given by the parameter $superscal$;
- out-of-order compute stages performing the actual computations in FUs, where N_{FU} specifies the number of FUs and RSs (FU$_1$ through FU$_{N_{FU}}$ and RS$_1$ through RS$_{N_{FU}}$);
- an in-order back-end (COM stage) in charge of committing instructions in program order using the ROB, where up to $superscal$ instructions can be committed.

The instructions proceed through the pipeline from left to right ($\rightarrow$), entering the pipeline at the IF stage and completing at the COM stage. Instructions are assigned their (future) actual FU in the ID stage and are then issued to the associated RS. When several FUs can perform the computation for one instruction, a random choice is made. The RSs and ROB track read-after-write data dependencies between instructions ($\leftrightarrow$) and thus allow to execute instructions out-of-order on the FUs. Results from FUs are immediately available ($\dashrightarrow$) and instructions may skip their RS and immediately proceed if the respective unit is not busy ($\cdots\rightarrow$). Consequently, (dependent) instructions can be executed back-to-back.

We formalized this pipeline using the TLA+ language [15]. TLA+ allows us to specify a transition system (TS) consisting of *states*, i.e., a set of *state variables*, a set of *initial states*, and a *next-state relation*, i.e., a transition relation producing a new state from a given state. In our case, the TS defines how the instructions of one trace proceed through the pipeline at the granularity of clock cycles. Notably, we model which instruction is processed by each of the hardware components depicted in Fig. 2 at any given instant.

The model depends on a set of input parameters (a in Table I), which enable to set the instruction sequence under analysis ($program$) and refine the characteristics of the architecture ($superscal$, N_{FU} and $IFLat$). Each instruction (within $program$) also contains execution constraints. It is associated with a set of operable FUs (FU affinities) as well as a set of possible latencies in the IF stage (modeling instruction cache hits/misses)[3] and in the FUs (e.g., modeling the data cache).

Our model makes use of some abstractions, i.e., the TS does not capture the entire architecture state that would appear on real hardware. As we only target timing effects that arise during execution, we do not model certain aspects of the architecture. For instance, opcodes and registers are not explicitly modeled, nor the actual computations of the FUs. More concretely, the state variables of our TS explicitly capture global time (variable $currCycle$), the status of the instructions in the trace ($prog$), and hardware components ($\_IF$, $\_ID$, $\_FU$, $\_RS$, and $\_COM$). The $prog$ variable is a record containing the remaining instructions to fetch (through the first member $prog.rest$) and the currently executing/completed instructions ($prog.exec$). The latter record member ($prog.exec$) captures the status of each instruction during execution (whether it has produced a result and whether/when it was committed). Additional variables track the actual assignment to FUs, the actual latencies, and other bookkeeping information. Those latter variables in combination with $prog.exec$ thus essentially implement the ROB.

The TS's next-state relation implements the fetch and issue logic in the front-end, the handling of the ROB, RSs, and FUs for the OoO computation, as well as the in-order commit in the back-end. The centerpiece of this function is Tomasulo's algorithm [14]—leaving out irrelevant functional aspects. We will now discuss how the definitions are mapped to this model.

[2]For instance, CVA6 (https://www.openhwgroup.org/) and BOOM (https://boom-core.org/).

[3]An instruction cache miss in one superscalar IF stage stalls all IF stages.

TABLE I: Summary of model parameters (a), state-independent helper functions (b), and helper state variables (c) used in the formalization of the definitions of TAs.

a	$program$	Input program with constraints about FU affinities and possible FU latencies	Sec. III-A
	$superscal$	Max. number of instructions fetched/issued/committed	Sec. III-A
	N_{FU}	Number of functional units	Sec. III-A
	$IFLat$	Set of possible latencies at IF	Sec. III-A
	$locFU$	Subset of FUs considered for component occupation	Sec. III-B3
b	$ProgDone(n)$	First n instructions were committed in both traces (Boolean)	Sec. III-B
	$ComTime(t,n)$	Time instant when n-th instruction was committed in trace t	Sec. III-B1
	$StepHeight(t,n)$	$ComTime(t,n) - ComTime(t,n-1)$; $ComTime(t,1)$ for $n = 1$	Sec. III-B2
	$FUusage(t,f)$	Number of cycles FU f is occupied in trace t	Sec. III-B3
c	$commonPre$	The two traces have not diverged at current time instant (Boolean)	Sec. III-B4
	$locWorst[t]$	Trace t represents the local worst case at current time instant (Boolean)	Sec. III-B4

B. Uniform Formalization of the Definitions of TAs

The parameterizable architecture template allows us to model the execution of a single trace, while tracking all of the trace's instructions as well as all the involved hardware components. In order to reason about TAs, we need (at least) two traces. We thus instantiate two *essentially* identical copies of the architecture model and program, where the two traces are restricted to the same instruction sequence, i.e., having the same set of dependencies, possible latencies and FU affinities. The differences between the two traces arise solely from variations in the *actual* latencies and FUs of instructions observed during execution. Instructions may advance through the pipeline according to their own actual latencies and assignments to FUs, the usage of hardware components by other instructions, and the dependencies among instructions.

This framework allows to implement the various definitions of TAs into a uniform formalization. More precisely, we define, in association with each definition, a procedure that decides whether a TA is signaled for the considered definition. These procedures are specified in the form of *predicates* (explained in detail further below) drawing on elements of the pipeline model. The predicates check the absence of TAs after the completion of each instruction in both traces, as a direct formalization of the key ideas and the assumptions introduced in Sec. II. In this regard, we rely on additional code and state variables, summarized in Table I. The table highlights state-independent helper functions (b), which operate on a trace's history, and helper state variables (c).

1) Intersections in Step Functions (TAInter): The key element of this definition (Sec. II-B) are commit events, which are tracked by additional state variables and accessible via the helper function $ComTime(t,n)$ (c.f. Table I). The related predicate is then directly expressed as:

$$\begin{aligned} NoTAInter \triangleq\ & \forall\, k \in 1 \,..\, progLen - 1 : \\ & \quad \forall\, n \in k+1 \,..\, progLen : \\ & \quad \wedge\ ProgDone(n) \\ & \quad \wedge\ ComTime(\alpha,\ k) \ < ComTime(\beta,\ k) \\ & \quad\ \implies ComTime(\alpha,\ n) \leq ComTime(\beta,\ n) \end{aligned}$$

As illustrated for the motivating example, a TA occurs when the intermittent order of the k-th instruction's commit (second conjunction) between the two traces (α and β) does not match the global execution time (rhs. of the implication)[4]—or inversely a TA is excluded when they always match, as expressed here.

Note that it is not necessary to consider the case where α and β switch positions in the formula, since TLC will explore all possible pairs of traces and thus will safely detect the anomaly anyway. This also applies to the subsequent formulas.

2) Step Height in Step Functions (TASteps): This definition (Sec. II-A) is very similar as the previous one. Here, a TA occurs when the order of the k-th instruction's *step heights* does not match the global execution time. Instead of the commit time, the $StepHeight(t,n)$, which is in fact derived from the commit time (cf. Table I), is used:

$$\begin{aligned} NoTASteps \triangleq\ & \forall\, k \in 1 \,..\, progLen - 1 : \\ & \quad \forall\, n \in k+1 \,..\, progLen : \\ & \quad \wedge\ ProgDone(n) \\ & \quad \wedge\ StepHeight(\alpha,\ k) < StepHeight(\beta,\ k) \\ & \quad\ \implies ComTime(\alpha,\ n) \leq ComTime(\beta,\ n) \end{aligned}$$

3) Component Occupation (TAComp): In order to express the predicate based on component occupation (Sec. II-C), additional state variables have to be added to the TLA+ specification that track the occupation of FUs, which is again accessible through a helper function $FUusage(t,f)$ (cf. Table I):

$$\begin{aligned} NoTAComp \triangleq\ & \text{LET } n \triangleq progLen \text{ IN} \\ & \quad \text{LET } usage(t) \triangleq \textstyle\sum_{f\, \in\, locFU} FUusage(t,\ f) \text{ IN} \\ & \quad \wedge\ ProgDone(n) \\ & \quad \wedge\ usage(\alpha) < usage(\beta) \\ & \quad\ \implies ComTime(\alpha,\ n) \leq ComTime(\beta,\ n) \end{aligned}$$

From the individual occupation obtained through $FUusage(t,f)$, the occupation of the supplied component is computed through summation. The FUs to consider in this component is provided as a model parameter $locFU$ (cf. Table I), since the choice of the partitions has little impact on the evaluation presented in Sec. IV. As before, the absence of TAs is stated when the relationship between the component occupation ($usage$ at the second conjunction) of the two traces is always the same as the global execution time (rhs. of the implication).

[4]In TLA+, logical connectors' alignment describes precedence; implication has the lowest precedence.

4) Instruction Locality ($TALoc$): This definition (Sec. II-D) is the most complex, since it is not based on simple numeric features as the other definitions. Firstly, TAs are associated with the time instant when the two traces *diverge*. As explained in Sec. II-D, we consider the two traces identical as long as the mapping of instructions to pipeline stages is identical. This is expressed through the state variable $commonPre$ (cf. Table I), which is initialized to $true$ and only reset to $false$ when this mapping differs. In our model, this is expressed through the terms $FUs[t][k]$ (and $IFs[t][k]$), which are records tracking the content of any FU (IF stage) k in trace t, from the $\_FU$ ($\_IF$) state variable of the trace. They give access to the assignment of instructions to FUs (member PC) and the current latency of the instruction on the FU (member $currLat$). Any divergence in the PC member causes $commonPre$ to be reset.

Once the traces are about to diverge, the occupation of the various pipeline stages have to be compared in order to determine which of the traces represents the local worst case. This is performed through the state variable $locWorst$:

```
$locWorst' = [t \in \{\alpha, \beta\} \mapsto \text{LET } o \triangleq \{\alpha, \beta\} \setminus t \text{ IN}$
    $\lor \neg commonPre \land locWorst[t]$
    $\lor \land commonPre$
        $\land \forall k \in 1 \,..\, N_{FU} : \forall kk \in 1 \,..\, N_{FU} :$
            $FUs[t][k].PC = FUs[o][kk].PC$
            $\implies FUs[t][k]'.currLat \geq FUs[o][kk]'.currLat$
        $\land$ Similar for $IFs$...
```

This TLA+ formula reassigns a new value to each member $locWorst[t]$ of this state variable (cf. Table I) for the next clock cycle—expressed by the prime operator in TLA+, i.e., the $'$ symbol. The formula then distinguishes two cases.

In the first case, the two traces have already diverged ($\neg commonPre$), $locWorst[t]$ then simply preserves its value for both traces, as indicated by the first disjunction.

The second case considers in particular the situation when the two traces are about to diverge, i.e., $commonPre$ is still $true$ but will be reset in the next cycle. At this moment, it is still possible to compare the occupation of the pipeline stages. We use the $FUs[t][k]'.currLat$ and $FUs[o][kk]'.currLat$ terms to detect the divergence in the next cycle (note the prime operator). In the next cycle, the considered instruction of one trace will have completed its computation in the considered FU. The $currLat$ value of that unit will thus be reset, and thus results in diverging values in the rhs. of the implication, i.e., the other trace becomes the local worst case. Note that we use independent FU indexes (k and kk) to allow for instructions that merely execute on another FU (see Sec. II-D). A similar check is also performed for the IF stage (as indicated by the shaded comment). Note that we do not consider the other hardware components for the comparison of latencies, since they do not exhibit variable latencies or are not suitable locations (Sec. II-D).

With these two additional state variables, it is possible to check for TAs using:

```
$NoTALoc \triangleq \text{LET } n \triangleq progLen \text{ IN}$
  $\land ProgDone(n)$
  $\land \neg locWorst[\alpha]$
    $\implies locWorst[\beta] \land ComTime(\beta, n) \geq ComTime(\alpha, n)$
```

The formula might appear surprising at first sight, as one might expect a formula where $locWorst[t]$ simply implies that t's global time is the largest. However, we have tried to state the definition as in the original paper [12]. Note that this may give quite different classifications of TAs, notably when both traces become local worst case due to opposing latency variations occurring at the same instant.

C. Formal Verification by Model Checking

In this paper, we do not intend to set up a (possibly efficient) procedure for detecting TAs over a wide range of programs, i.e., from a specific (reliable) definition. We use model checking in order to assess the various definitions of TAs discussed in Sec. II. We invoke TLC, the TLA+ model checker, on the parameterizable model, in order to explore variations of otherwise identical traces while evaluating the various predicates, as potential culprits for TAs. Verifying the absence/presence of TAs in this way helps us to find inconsistent scenarios, e.g., where some definitions identify a TA for an instruction sequence while others do not.

The state space that is to be explored mainly depends on the *program* and *IFLat* parameters, namely the length of the input program, the program dependencies, and the specified execution variability through possible latencies and FU affinities. The depth of the state space is approximately the program length and the breadth is fully determined by the set of initial states. Denoting as $FULat$ the set of possible FU latencies (imposed in fact as program constraints), checking the absence of TAs for *all* the input programs of length N would require exploring all possible dependencies ($\Pi_{k=0}^{N} 2^k \approx 2^{N^2/2}$), as well as all variations of IF and FU latencies for each instruction in each trace ($|IFLat|^{2N}$ and $|FULat|^{2N}$) and all FU affinities (${N_{FU}}^{2N}$). Those terms essentially multiply and quickly result in a very large state space.

However, we do not need to consider all programs. We merely aim at getting several scenarios, i.e., pairs of execution traces, that expose the contradictions among the definitions and their limitations. Those scenarios may differ from the input program and/or architectural parameters, e.g., $superscal$ and N_{FU}. Such scenarios are derived as *counterexamples* for specific – violated – properties. These properties are *invariants* expressing, for instance, that some definitions *always* make consistent statements about TAs (for fixed values of the parameters). We systematically *analyze* the obtained counterexamples to confront them to the intuitive understanding and then set up an invariant expressing another inconsistent scenario, e.g., a contradiction between another pair of definitions. Our probation methodology starts from the motivating example (in Fig. 1d), then progressively proceeds through more elaborated variations in order to obtain convincing counterexamples that are short, easy to understand, and still illustrate a relevant shortcoming of at least one of the definitions. We supply program

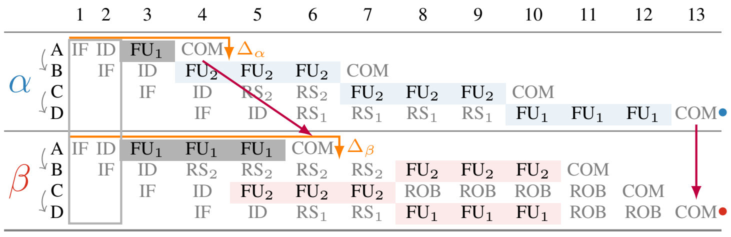


Fig. 3: Importance of structural aspects (Sec. IV-A)

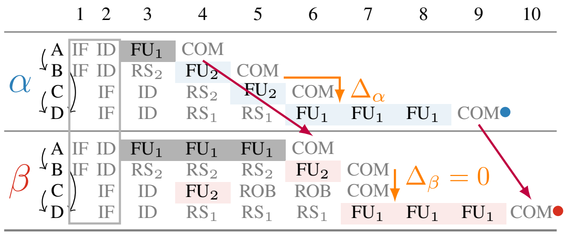


Fig. 4: Step height and execution order (Sec. IV-B)

portions made of a few instructions and we perform progressive variations on the dependencies and on the constraints restricting the possible FU latencies and affinities. In most examples, we only vary the FU latencies (since this is enough to demonstrate the short-comings). In some cases, we also allow variations on the IF latencies to highlight specific shortcomings. Consequently, we do not face state space explosion and the worst complexity in our execution scenarios exposed in the next section is illustrated by a counterexample (Sec. IV-E) requiring TLC to execute for 6 seconds and explore about 1,000 states. TLC provides counterexamples by enumerating the sequence of states that represent the execution scenario. Its output can be parsed automatically in order to obtain a visual illustration, e.g., the basis of all examples in Sec. IV.

IV. Short-Comings of Existing Definitions

In this section, we present the (counter)examples that we have retained for illustrating the short-comings of the definitions.

A. *Importance of Structural Aspects*

This example shows that taking into account the structure of the *whole* pipeline and its *architectural features* is important to identify TAs. Let us resume the example of Fig. 1 ($N_{FU} = 2$, see Sec. III-A) and modify the *superscal* parameter (from 2 to $superscal = 1$) to allow at most one instruction per cycle in the in-order stages of our model. We only present the detailed table-based execution traces, which nevertheless contain all the information, and not the other representations, i.e., time plots. The execution traces, in both scenarios, are given in Fig. 3. Trace β takes longer compared to the dual-issue pipeline of Fig. 1d and all of the formal definitions correctly reflect the consequent intuitive absence of TAs. Though the definitions agree here, this first example confirms that: (i) the common scheduling diagrams (such as Fig. 1b), almost exclusively used in the literature, are *not* sufficient to study TAs; (ii) *executable* models of hardware platforms and program executions are beneficial for the concrete assessment of TAs.

B. *Step Height and Execution Order*

Both $TASteps$ and $TAInter$ require the completion of entire instructions, namely the observation of *commit events*. The slightly modified example of Fig. 4 shows that the step height of a specific instruction is likely to present TAs in unexpected situations. It is primarily obtained as a violation of the invariant $NoTAInter \implies NoTASteps$, i.e., the definition $TASteps$ states a TA though $TAInter$ does not. There is no clear intuitive TA, since trace β with the 3-cycle latency in FU_1 leads to the WCET. Yet, (only) $TASteps$ indicates a TA. Indeed, the latency for instruction C in trace β is zero, because it is committed at the same time as B. While any inversion in cumulative execution times necessarily originates from variations in step heights at some points, this example shows that the converse is not true. The step-height metric is too coarse-grained since values cannot be negative (due to in-order commit). Consequently, it is not adequate to define and ultimately reason about TAs in terms of step heights.

C. *Intersections and Execution Order*

So far, we discarded only one definition among those based on commit events. The other definition, $TAInter$ seems, so far, to be more adequate. However, this new example shows that surprising TAs arise here too. It is obtained with the violation of the invariant: $NoTALoc \implies NoTAInter$, where $TALoc$ is taken as an initial postulate. In the previous examples, no instruction memory accesses are performed (plausible with an instruction scratchpad). If instead we consider an instruction cache, a cache miss might increase the fetch delay. In Fig. 5, we present a pair of program executions with no variations in FUs, but a possible instruction cache miss for the last two instructions. Here again, there is no clear counter-intuitive TA, since the WCET is indeed given by the worst-case scenario,

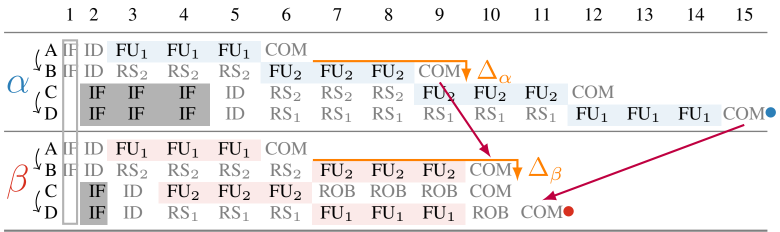


Fig. 5: Intersections and execution order (Sec. IV-C)

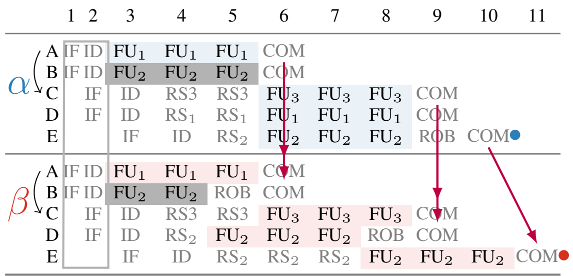


Fig. 6: Commit events and relevance of locality (IV-D)

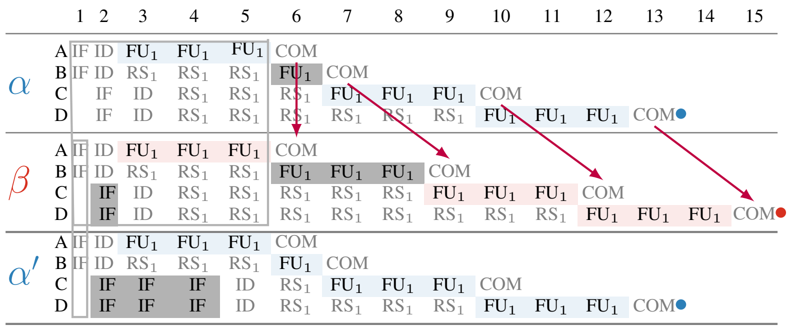


Fig. 7: Comparing occupation of locations for locality (Sec. IV-E)

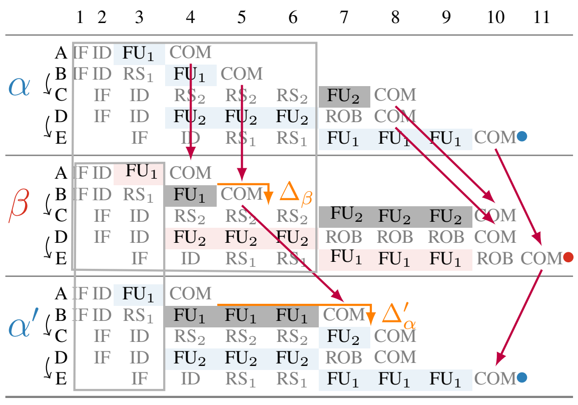


Fig. 8: Component occupation (Sec. IV-F)

i.e., the instruction cache miss (trace α). This is confirmed by TALoc, which does not signal a TA (see the gray box and occupation). In the case of $\mathit{TAInter}$ (and $\mathit{TASteps}$), there is an evident inversion and thus a TA. There is however a particularly surprising timing effect. The favorable case, i.e., the cache hit in trace β, entails a scheduling similar to the one of traces β in the previous examples, namely instruction C starting its computation before instruction B, delaying the commit of B. Intuitively, the global execution of the unfavorable case α needs 4 additional cycles, whereas its cache miss shows a 2-cycle difference compared to the cache hit. This is an *amplification* effect that shows that both counter-intuitive and amplification TAs are closely related.

D. Deficiencies of Commit Events and Relevance of Locality

Showing unspecified behaviors is not the only shortcoming of $\mathit{TAInter}$, this formulation is also unable to detect all TAs. The example from Fig. 6 shows that its high-level granularity based on commit events is insufficient, a finer control of pipeline executions (e.g., as in TAComp or TALoc) is required. This example is based on the execution of a program with five instructions on an architecture with $N_{FU} = 3$. It is derived from the violation of the invariant: $(\mathit{NoTAInter} \lor \mathit{NoTASteps}) \implies \mathit{NoTALoc}$, assuming here that TALoc is reliable for detecting TAs. Instruction B has a variable latency in FU_2 and instruction D can execute either on FU_1 or FU_2. The two execution scenarios in the figure show choices of different FUs after a variable latency of instruction B (plausible if instructions are preferably issued to FUs that are not busy). The seemingly most favorable case, i.e., 2 cycles in FU_2 (trace β), eventually leads to the global worst case, which is intuitively a TA. Yet, only the last instruction differs in terms of commit events and hence the definitions based on commit events, $\mathit{TAInter}$ and $\mathit{TASteps}$, are unable to detect it. The definition TALoc has the shorter trace α as local worst-case (and not the trace β) because of instruction B, which does correspond to a TA. Similarly, TAComp could detect a TA, though depending on a hardware partitioning. For instance, with the (sub)set $\mathit{locFU} = \{FU_1, FU_2, FU_3\}$ (all highlighted cells), the TA is detected, because of the way FU_2 is used.

E. Concern about Local Occupation for Applying Locality

We mentioned in Sec. II that a major concern of the original work defining TALoc is how to reason about the *local* occupation of FUs. However, the example in Fig. 7 shows that the concern remains even with the hypotheses added in the background section. All instructions execute on FU_1 and, considering only the traces α and β, trace β is the local worst-case and there is no TA (whatever the definition). Let us assume that the hardware model brings up (only) a third trace α', in which the last two instructions experience instruction cache misses. α' and β are concretely derived from the violation of the invariant: $(\mathit{NoTAComp} \lor \mathit{NoTAInter}) \implies \mathit{NoTALoc}$.[5] The local worst-case is now α' and TALoc identifies a TA, due to the variation in instruction fetching. Note that with TALoc we cannot properly compare the local occupation of FU_1 by instruction B for traces β and α', since these traces have diverged when B starts its computation on FU_1. Actually, the variation in fetching (α vs α') is independent of the one in the FUs (and does not impact the scheduling on FUs). It is however clear that the verification based on commit events still states the absence of TAs as well as when applying TAComp. Comparing local resource usage in this way is unreliable under more than one source of variations.

F. Issue with Component Occupation

We already showed that the arbitrary choice of relevant FUs for TAComp has an impact on stating the absence of TAs (Sec. II). This example shows that TAComp is not stable even with a preset hardware partitioning. From Fig. 8, we consider two independent variations in the computation latency of two instructions B and C. Firstly, we consider the traces α and β and we fix, for TAComp, the subset of FUs as $\mathit{locFU} = \{FU_1\}$. Under TAComp these traces do not present a TA since the component occupation is 5 for both α and β—btw., the other TA definitions yield the same answer. However, the slightly modified trace α' emphasizes the previously introduced issue on comparing local resource usage, while TAComp cannot address it: α' and β are merely derived

[5] α and β are obtained from a property stating the absence of TAs from all definitions.

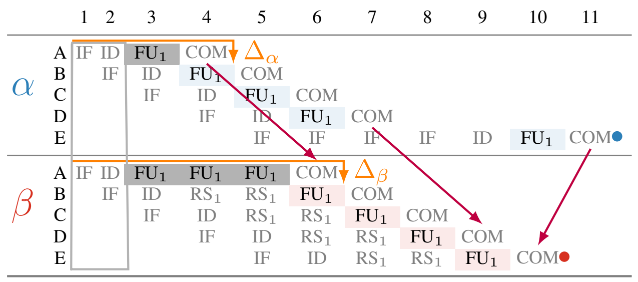


Fig. 9: End of the instruction sequence (Sec. IV-G)

from the violation of the invariant: $NoTAComp \wedge NoTALoc$.[5] When we consider the α' and β traces, both $TALoc$ and $TAComp$ show the presence of a TA, since the component occupation in trace α' is 7. In any case, it is difficult to interpret the results of $TAComp$ since this definition gives absolutely no information whether a certain scenario is identified as a TA. Actually, wrt. α' and β, all definitions state a TA incriminating instruction B (because of its commit event or its FU latency). Yet, B cannot be the *cause* of a TA, since its execution variation (i.e., the latency in FU_1) is completely hidden by the execution of D. Specifically, instruction D is allowed to immediately start its computation in FU_2, so the computation of B in FU_1 does not alter the FU scheduling and C always starts its computation in cycle 7 (and from that point, the single variation in FU_2 has no surprising effect). Intuitively, these independent variations do not generate a TA, however no definition is able to separate the effects of the two variations (e.g., starting in cycle 7).

G. *Issue about the End of the Instruction Sequence*

Up to now we focused on the main differences between the proposed definitions, namely the notion of local variations whose comparisons define what a favorable case is. The global comparisons are simpler to establish since these are based on the end of a certain *last* instruction; in our model, the end of an instruction is clearly defined by its commit event. All definitions rely on an instruction *sequence* within a program but the choice of this sequence is essential to properly assess the TAs. Let us consider the simple example with a single FU and a single FU latency variation, in Fig. 9. This counterexample is derived from an invariant involving copies of the predicates of the definitions in which one can specify as a parameter the last instruction to consider instead of the program length. If we do not consider the last instruction, the single variation drives the whole execution (with instruction A in FU_1) and all definitions confirm the absence of a TA. However, no definition could accommodate the fact that the last instruction(s) could add irrelevant extra cycles. If we consider the complete sequence (including the instruction E), all definitions state a TA. It is interesting to remark that E is fully independent of the other instructions and no definition is able to capture the variation of this last instruction.

V. Further Related Work

In this paper, we aim at focusing on the core of the various definitions of TAs. Yet, other questions deserve to be examined in greater detail. First, we consider that TAs manifest essentially in concrete hardware, as they were originally observed [5] and as it was later supported [9]. Eisinger et al. [10] actually define TAs from an *abstract* reference execution scenario. Reineke et al. [12] also rely on an abstract behavior. Consequently, their underlying TS cannot be a faithful model of the actual hardware. It is unclear how to obtain suitable abstractions in practice. These abstractions may have a strong impact on the study of TAs, independently of the pipeline granularity. Then, the definitions differ in which trace variations may trigger a TA. In our examples, we did not seek to unduly justify why we focused on those specific traces, since we consider that any execution pair is suitable for defining TAs. Eisinger et al. [10] successively compare traces to the reference behavior that might correspond to the assumptions of a particular WCET analysis. They thus verify a necessary (not sufficient besides) condition for the validity of an execution-time upper bound, instead of the strict absence of TAs. Similarly, Reineke et al. [12] target the case always making the worst local decisions against all other traces, which is logically related to the verification of a specific WCET analysis. Cassez et al. [9] target pairs involving the actual WCET. The TAs targeted by these – essentially different – approaches impacting the WCET could be generically called *strong* [10]. Some definitions are unclear about how to deal properly with more than two traces simultaneously. Finally, it is unclear what (initial) execution context to consider when concretely applying the definitions. Our model assumes that the pipeline is initially empty, but real architectures may execute instructions in richer contexts.

In a larger scale, TAs are essential to numerous investigations that intend to detect them or to study their effects. Eisinger et al. [10] also used model checking to detect TAs automatically, however (besides the abstract reference behavior) without providing details about their formal models and with a too coarse instruction-level granularity ($TAInter$, Sec. II-B). From the previous section, we also consider the pipeline-stage level retained by Engblom [16] and later Reineke et al. [12] the right granularity. Asavoae et al. presented a first attempt to make Reineke et al.'s definition executable and detect TAs by model checking [17], [18]. This work assumes this definition and shows how it can be auspiciously integrated into an automatic tool. Hahn et al. analyzed amplification TAs and their link with compositionality [19], however they did not provide a formal definition. Hahn and Reineke continued this work with the development of a pipeline designed to be free from TAs [20]. They based their analysis on a monotonicity property resting on a progress notion [21], as a sufficient condition for the absence of TAs, and did not aim at providing a formal *definition* of TAs. Moreover, though these concepts are intuitive, they are specialized for classical in-order pipelines and it seems difficult to transpose them to an OoO context. Jan et al. used model checking in order to prove the absence of amplification TAs in predictable pipelines and compare their hardware approach to avoid such TAs [22]. We extended this work with a more complex case study showing amplification TAs and thus provide abstractions to accelerate their detection [23]. This approach is based on a prerequisite for TAs (in a particular situation) and does not aim at assessing the assumed definition.

VI. Discussion and Conclusion

The *formal* and *executable* models that we have developped are essential to evaluate the existing definitions of TAs. Many definitions of TAs found in the literature are dissociated from concrete hardware architectures. As a consequence, these definitions are mainly theoretical and cannot be integrated *as is* into automatic tools. We thus specified precise assumptions to make them *all* applicable to an *automatic* detection of TAs on standard OoO architectures. We notably showed that common FU scheduling diagrams are not sufficient to reason about TAs and that structural aspects must be taken into account.

Then, the systematic investigation of TAs showed that the definitions often lead to contradictory statements about TAs. The various exposed execution scenarios represent different situations (e.g., programs) reflecting plausible executions in an OoO pipeline and showing specific limitations of the definitions. Thus, even under a precise evaluation framework, it is impossible to fix conditions under which a given definition behaves consistently and could serve as a reference in place of the intuitive interpretation: **none of the existing definitions of TAs dominate others**.[6] Moreover, we carefully analyzed the counterexamples to clarify the reasons why some definitions state a TA or not: **no definition is reliable when put to the test on an OoO pipeline**, i.e., none is always consistent with the intuitive understanding widespread in the community. Consequently, a precise formal definition of TAs is still needed.

The examples show that the contradictions of the previously proposed formal definitions, corroborated with the intuitive understanding, stem from a common deficiency: the notion of **causality**. These definitions are based on the presumed relation between local variations and global execution times. Yet, nothing ensures that a variation of a global execution time is *due to* the variation of an assessed local execution time, even with a single local variation. Such a causality link is however central in the intuitive perception of a TA. We observed that the definitions based on commit events can be easily manipulated by shifting the moment when a certain instruction ends, independently of surprising timing effects. The definitions based on components/locations could target local variations that would not be intuitively considered determining, as soon as an instruction sequence entails two (local) variation sources. The omission of causality is thus the main defect shared by all formal definitions.

We exemplified short-comings of the existing definitions of TAs on a standard architecture and on simple, short instruction sequences. These definitions would be incomplete a fortiori on a more complex architecture and larger programs. Once a precise definition established, TAs should be detected with appropriate procedures in the scope of timing analysis, on programs with multiple long execution paths. In particular, the concern about the relevant end of a sequence for identifying TAs turns into defining an observing interval. Here again, the notion of causality should prevent from arbitrarily slicing a program before studying the timing effects entailed by its execution. All the TAs identified in the previous section are interpreted through scheduling of instructions and depict the most commonly described class of TAs in the literature. However, TAs could also arise from speculation or cache effects [12], even in in-order pipelines [7]. The notion of causality will be all the more relevant in these cases. Our ongoing work is to integrate the crucial notion of causality – and its various incarnations – into a *precise* and *practical* formal definition of TAs, and then to develop efficient detection procedures that allow to prove the absence/presence of TAs in the execution of a program on a given processor architecture.

[6]The same holds when considering $TAInter$ and $TAComp$ (by the same authors) supplementary definitions.

References

[1] 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 problemoverview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, May 2008.

[2] S. Law and I. Bate, "Achieving appropriate test coverage for reliable measurement-based timing analysis," in *ECRTS*, 2016.

[3] F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega, "Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey," *ACM Comput. Surv.*, 2019.

[4] R. Davis and L. Cucu-Grosjean, "A survey of probabilistic timing analysis techniques for real-time systems," *LITES*, 2019.

[5] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Real-Time Systems Symposium*, 1999.

[6] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder, "Principles of timing anomalies in superscalar processors," in *QSIC*, 2005.

[7] G. Gebhard, "Timing Anomalies Reloaded," in *WCET*, 2010.

[8] R. Kirner, A. Kadlec, and P. Puschner, "Worst-case execution time analysis for processors showing timing anomalies," TU Wien, Tech. Rep., 2009.

[9] F. Cassez, R. R. Hansen, and M. C. Olesen, "What is a Timing Anomaly?" in *WCET*, 2012.

[10] J. Eisinger, I. Polian, B. Becker, S. Thesing, R. Wilhelm, and A. Metzner, "Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis," in *DDECS*, 2006.

[11] R. Kirner, A. Kadlec, and P. Puschner, "Precise worst-case execution time analysis for processors with timing anomalies," in *ECRTS*, 07 2009.

[12] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A Definition and Classification of Timing Anomalies," in *WCET*, 2006.

[13] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for wcet analysis," *Real-Time Systems*, 2006.

[14] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, 1967.

[15] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[16] J. Engblom, "Processor pipelines and static worst-case execution time analysis," Ph.D. dissertation, 04 2009.

[17] M. Asavoae, B. B. Hedia, and M. Jan, "Formal Executable Models for Automatic Detection of Timing Anomalies," in *WCET*, 2018.

[18] M. Asavoae, M. Jan, and B. Ben Hedia, "Formal modeling and verification for timing predictability," in *ERTS*, 2020.

[19] S. Hahn, M. Jacobs, and J. Reineke, "Enabling compositionality for multicore timing analysis," in *RTNS*, 2016.

[20] S. Hahn and J. Reineke, “Design and analysis of sic: A provably timing-predictable pipelined processor core,” in *RTSS*, 2018.

[21] S. Hahn, J. Reineke, and R. Wilhelm, “Toward compact abstractions for processor pipelines,” in *Correct System Design*, 2015.

[22] M. Jan, M. Asavoae, M. Schoeberl, and E. A. Lee, “Formal semantics of predictable pipelines: a comparative study,” in *ASP-DAC*, 2020.

[23] B. Binder, M. Asavoae, F. Brandner, B. B. Hedia, and M. Jan, “Scalable detection of amplification timing anomalies for the superscalar tricore architecture,” in *FMICS*, 2020.