The Role of Causality in a Formal Definition of Timing Anomalies

Benjamin Binder, Mihail Asavoae, Florian Brandner, Belgacem Ben Hedia, Mathieu Jan

▶ To cite this version:

Benjamin Binder, Mihail Asavoae, Florian Brandner, Belgacem Ben Hedia, Mathieu Jan. The Role of Causality in a Formal Definition of Timing Anomalies. 2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug 2022, Taipei, Taiwan. pp.91-102, 10.1109/RTCSA55878.2022.00016 . hal-03867187

HAL Id: hal-03867187

https://telecom-paris.hal.science/hal-03867187

Submitted on 8 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Role of Causality in a Formal Definition of Timing Anomalies

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

***Abstract*—Intuitively, a counter-intuitive timing anomaly manifests when a locally faster execution becomes globally slower. While the presence of such timing anomalies threatens the soundness and/or scalability of timing analyses, tools to systematically detect them do not exist. The main reason lies in the absence of a definition of counter-intuitive timing anomalies that establishes relations between local and global timing effects. In this paper, we address these relations through an important concept, that of causality, which we further use to revise the formalization of counter-intuitive timing anomalies. We also propose a specialized instance of the notions to implement a detection procedure for out-of-order pipelines.**

***Index Terms*—Timing Anomalies, Formal Modeling, Out-of-Order Pipeline**

I. Introduction

Reasoning about real-time systems means reasoning about their *timing* behavior through dedicated timing analyses, for example WCET analysis. Undesired timing phenomena, called *Timing Anomalies* (TAs), can manifest, threatening the soundness of such timing analyses. We can distinguish amplification anomalies, important for compositionality [1]–[3], and counter-intuitive anomalies. In this work, we consider counter-intuitive TAs only, which could be explained using a pair of execution traces executing the same program trace, i.e., the same input program and data, but starting from distinct initial hardware states. A counter-intuitive TA occurs when one execution is locally faster (e.g., cache hit vs. miss) but globally slower than the other one. The local timing contribution of an instruction in one trace is often called a latency that entails a variation wrt. the other trace, which *triggers* the TA.

WCET analysis computes execution time bounds of a given program, running on an underlying hardware. Static WCET analyses use abstractions to compute an over-approximation of the possible hardware states that may appear during the program execution [4]. On complex hardware, this inevitably leads to state explosion and thus, it would be desirable if abstract states could be pruned during the analysis. As pruning should also preserve the soundness of the WCET analysis, the source of TAs is subject to two interpretations.

A first interpretation considers TAs to be an artifact of the analysis in combination with pruning [5], [6]. A TA is identified for a given abstraction when the WCET bound obtained with pruning is smaller than without pruning. Note that under this interpretation, TAs may be assumed even when the WCET bound obtained with pruning is safe wrt. the concrete hardware. TAs, however, were originally *observed* in a real processor [7], outside of any analysis framework. This leads to another interpretation, where TAs originate from the hardware itself [8], [9]. Pruning during WCET analysis then becomes unsafe when the actual WCET is larger than the WCET bound obtained from the analysis. *Studying TAs then consists in determining whether the hardware fulfills the underlying assumptions of the analysis.*

This second interpretation is also applicable outside of traditional static WCET analysis. Notably, TAs may be problematic when trying to bound the impact of perturbations that may occur during the execution of a program. Preemptions or interrupts would be a typical example of such situations. The perturbations may lead to new hardware states that would not occur during an execution of the program in isolation, e.g., an instruction cache miss might occur even for two successive instructions on the same cache line. WCET analyzers would probably fail in exhaustively exploring the possible states. Besides, TAs jeopardize other timing-analysis methods partially based on concrete executions, such as measurement-based analysis [10] and probabilistic analysis [11]. The impact of TAs thus has to be studied and understood independently from the WCET analysis technique itself. In this work, we adhere to the second, hardware-centric interpretation.

Several formal definitions of counter-intuitive TAs were proposed in the literature [6], [8], [9], [12]–[14]. However, recent work [15] has shown that none of these definitions is able to correctly capture TAs on an Out-of-Order (OoO) pipeline. These definitions share several issues. First, nearly all are based on hardware models that remain theoretical concepts, without a clear relation to concrete hardware. As a consequence, they are often not implemented as TA-detection procedures or in any other practical setting. Second, they lack a way to correlate the local timing variations and their impact on global execution time. Again, this shortcoming could explain the absence of tool support to reason about TAs.

We propose a framework that lays the groundwork for the identification of TAs—unambiguously applicable to a concrete architecture and independent of the WCET analysis method.

1) We propose a formalization of TAs based on the notion of *causality*, which restricts the scope of a variation to the trace portion where it determines the timing behavior.
2) We instantiate this formalism on a *well-specified hardware-architecture model* representing an OoO pipeline, with all the necessary information for the TA-detection procedure that we have implemented.

3) We evaluate the instantiation of our formalism on the OoO pipeline wrt. false positives and faithfulness of the representation of scheduling effects established as *TA patterns*.

4) We identify a new problem, related to the *composition* of multiple variations, since our accurate formalization of TAs exposes more complicated scenarios on the traces under consideration. We consistently represent multiple variations individually, which allows for tackling this open problem that we plan to address in future work.

The rest of this paper is organized as follows. We address related work in more detail in Sec. II. In Sec. III, we describe a comprehensive example to motivate our overall approach. We informally introduce how we interpret the notion of causality, which is then used, in Sec. IV, to formalize TAs. Sec. V details the correctness arguments. In Sec. VI, we present the application of our detection procedure on examples, illustrating the capabilities of our approach, before concluding in Sec. VII.

II. Related Work

The crucial notion of causality is missing in all the proposed formal definitions of TAs, in particular in the case of multiple variations. The semi-formal definitions of TAs by Lundqvist and Stenström [7] and Wenzel et al. [5] explicitly assume that a single variable latency affects the trace comparison, whereas no later formal definition restricts the way that traces may differ from each other. In our work, we generalize it to possibly multiple variations. Some definitions are based only on the commit events [8], [9], [12], [14], representing the instants where instructions leave the pipeline, to define latencies and variations. For instance, Gebhard [8] defines latencies as the timing gaps between the commit events of two successive instructions. These definitions are unsafe: they are too coarse-grained and omit relevant events in-between instruction commits, which may *hide* TAs and thus lead to inconsistent verdicts [15]. Only the definition by Kirner et al. [13] is not based on variations identified by instructions, but on a global favorable (i.e., lower) total utilization of certain resources that constitute a *hardware partition*. However, this definition does not state how to determine relevant resources in order to formally define partitions. The detection of TAs is not conclusive, e.g., one instruction could increase its use of a resource while another instruction decreases its use of the same resource, leading to the same total utilization in both traces [15]. Finally, the definition by Reineke et al. [6] is based on finer-grained comparisons of traces, however with specific limitations that are detailed below (cf. Sec. III). A relaxed version [16] focuses only on the instants when instructions are fetched in order to define latencies, thus with the same granularity as the aforementioned coarse-grained definitions.

Eisinger et al. [14] apply model checking on an OoO processor with OoO commit to identify TAs wrt. a WCET *analysis abstraction* of the processor. The formulation is only based on commit events and does not integrate causality. Similarly, Asavoae et al. [17] apply the definition by Reineke et al. [6] in a model checker in order to detect TAs on an OoO processor—thus inheriting the limitations of the underlying definition. We also have used model checking in previous work to highlight issues of existing definitions [15]. This work shows the relevance of causality with an intuitive concept but does not propose any novel form of definition nor indicate how to concretely derive causal relationships.

Specific cores based on hardware counter-measures have been designed in order to remove TAs. The Patmos [18] and SIC [19] cores target particular TAs occurring due to memory interference. These cores are based on simple in-order pipelines and it is unclear how to extend these results to common OoO pipelines. MINOTAuR [20] extends the framework of SIC to a more complex architecture, which allows speculative execution and can execute independent instructions out-of-order (stalling decoding on dependencies and respecting program order on functional units). Similarly, Vicuna [21] is a timing-predictable vector co-processor. All these cores prevent execution scenarios that are known to entail TAs, but no definition of TAs per se is provided.

Our approach for modeling timing dependencies in a pipelined processor is similar to that of Li et al. [22]. They use *Execution Graphs* (EG) that model the timing semantics of an OoO pipeline through events, latencies and the imposed order between events, in order to provide analytical WCET estimates. We need to capture causality emerging from *dynamic* scheduling effects, which may trigger TAs. We thus focus on the execution of *specific* traces. We also need runtime information such as the actual instant of each event and the actual order imposed by contention (the EG uses *undirected* arcs in this case). Bai et al. extend the EG with a more compact but equivalent symbolic data structure called Execution Decision Diagram [23]. Hahn et al. [1] use *microarchitectural execution graphs* to represent possible durations between events of interest. However, they focus on abstract states that do not allow distinguishing the effects of individual instructions needed to identify causal relationships.

III. Motivation

We consider a sequence of instructions A to E with data dependencies, and a simple OoO pipeline (see Fig. 2) with three Functional Units (FUs). Fig. 1 presents four traces, α, β, α', and β', corresponding to different pipelined executions of this sequence, from distinct initial hardware states yet the same input data. These traces may exhibit a local variation wrt. the use of FU_1 by instruction A. This particular instruction executes 1 cycle on FU_1 in α/α' (e.g., data-cache hit) and 3 in β/β' (e.g., data-cache miss). These traces may also exhibit a local variation wrt. the use of the IF stage by instruction E, which requires 1 cycle in α and β (e.g., instruction-cache hit) and 3 cycles in α' and β' (e.g., instruction-cache miss).

The definition of TAs by Reineke et al. [6] is the most appropriate wrt. the granularity of variations (cf. Sec. II), since the underlying latencies are based on the local occupation of a pipeline resource by an instruction. This definition assumes a hardware abstraction and consequently, a static analysis to compute abstract hardware states. This abstraction can be arbitrarily close to the concrete architecture. We thus

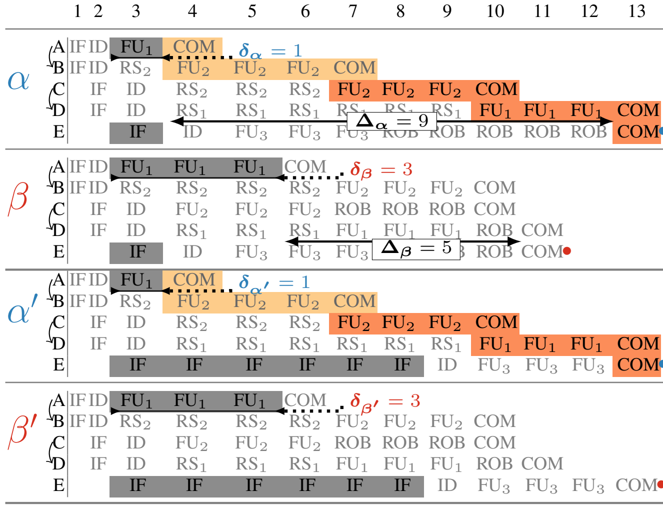


Fig. 1: Execution of a program with data dependencies (↰) on an OoO processor. Traces α/α' vs. β/β' exhibit a local variation wrt. FU_1 for A (▬). α/β vs. α'/β' exhibit a local variation wrt. IF for E (▬). β' is the local worst-case path according to the definition by Reineke et al. [6], which indicates the absence of TAs. α/α' vs. β/β', however, constitute a well-established example of a TA [5], [6], [13], [17]. δ, Δ, and the colored cells are relevant for our definition (cf. Sec. IV-F).

assume an abstraction that explores all (concrete) states and we consider that an abstract state maps each pipeline resource to the (potential) processed instruction at a given instant and to the associated latency. Then, whenever two traces share the same prefix and then diverge due to a local variation, this definition identifies the variation with the larger latency as a local worst-case. A local worst-case path is a trace that follows all the local worst-cases (i.e., for any other trace and any variation). Consequently, this definition identifies a TA when the WCET bound is not derived from a local worst-case path. During the analysis, if there is no TA, it is safe to prune the abstract states that open up a non-local worst-case path.

The four traces in Fig. 1 share the same prefix comprised of the first two cycles and then diverge due to two distinct, simultaneous variations in cycle 3, i.e., the use of FU_1 by A and of IF by E. Both variations identify β' as a local worst-case wrt. any other trace, thus this trace is a local worst-case path. Since this local worst-case path results in the global WCET, this definition does not signal a TA. However, a WCET analyzer could determine that trace β' is infeasible (e.g., the case with two cache misses is excluded) and thus, this path should be excluded (i.e., for a tighter WCET bound). More precisely, the abstract state that opens up β' in cycle 3 should be pruned from the state space. A direct consequence of this pruning is that, in order to remain sound when applied, this definition should also address the remaining traces. It does not because it follows only the local worst-case paths.

The pruning under the local worst-case path leaves us with the traces α, β, and α'. Let us inspect them closer. We note that instruction E does not impact the scheduling of the other (preceding) instructions because E has no data dependency and is the only instruction to use FU_3. Moreover, E does not impact the global execution time, which is, in fact, determined by the execution of instructions A to D (i.e., the same in traces α and α'). We also note that the variation in the use of FU_1 by A, in traces α/α' as opposed to β, affects the scheduling of the instructions B to D. This variation is favorable (i.e., A has a shorter latency) in α/α', and leads to a global slowdown wrt. β, notably for the commit of D and E. This case is a traditional TA pattern often found in the literature [5], [6], [13], [17].

The definition by Reineke et al. [6] is unable to identify a local worst-case path among the remaining traces α, α', and β. Both variations are favorable for trace α, thus this trace cannot be a local worst-case path. Traces β and α' mutually prevent each other from being identified as a local worst-case path, since each trace has a variation that constitutes a local worst-case that appears precisely when the traces diverge. However, the local worst-case related to the use of FU_1 by A should serve as a basis to actually identify trace β as the local worst-case path. As a consequence, this definition is limited for comparing traces, thus for consistently reasoning about TAs.

We argue that a definition of TAs should be able to identify individual variations and to check whether these variations could actually *determine* global slowdowns. Moreover, it should be able to identify chains of events from any favorable variation, defining trace portions of interest (▬/▬), later called *causal regions*. As such, a TA would be stated wrt. a trace if a slowdown is observed (▬) in the causal region. A formal definition that is able to identify variations and causal regions, to work under less restrictive assumptions (i.e., the existence of a static analysis to compute abstract hardware states) and to systematically discriminate between traces is introduced next.

IV. Formal Definition of Timing Anomalies

In this section, we gradually develop a formal definition of counter-intuitive TAs. We rely on two input **traces** derived from a transition system, from which we define **events** at the granularity of pipeline resources. The events represent the acquisition and the release of resources at some instant. From these events, we define an *Event Time-Dependence Graph* (ETDG) for each trace, whose arcs capture timing dependencies expressing the fact that the source event imposes a minimal duration before which another destination event cannot occur. The interval between the instants of the acquisition and the release of a resource by the same instruction defines a **latency**. From both graphs, we define **(favorable) variations** in the use of resources. Then, we introduce a *Causality Graph* (CG), i.e., a sub-graph of the ETDG where only the arcs that *exactly* and unambiguously explain the instant of the destination event remain. We then introduce the **causal region** of a favorable variation, as a sub-graph of the CG, to define the scope in which the variation determines the timing of other events. Finally, we combine all these elements to precisely capture counter-intuitive TAs triggered by the variation.

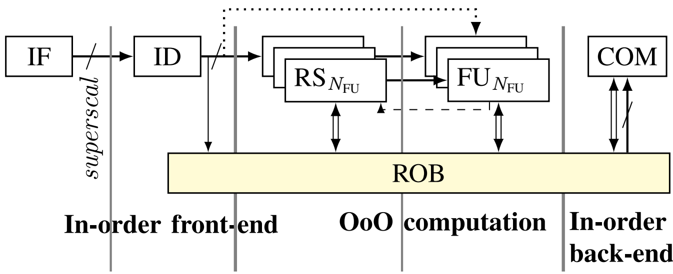


Fig. 2: OoO-pipeline model with N_{FU} functional units, fetching and committing at most $superscal$ instructions per cycle.

We first define general concepts and then provide concrete instantiations for our case study, a representative OoO-architecture model. We focus on this precise microarchitectural model, providing modeling details and an actual implementation of a TA-identification procedure.

A. Execution Model for Timing Anomalies

We first define execution traces, i.e., how the hardware executes a given sequence of instructions. The targeted hardware is a pipeline that may perform computations out-of-order. We only assume that the computed results are committed in-order—which is the case for modern processors.

Definition 1: Execution Traces — The set $\mathcal{S}$ of *execution traces* of a hardware model, represented by a transition system (TS), consists of all finite sequences of instructions executed by the TS, from any possible initial state.

Instantiation 1: Hardware Model & Execution Traces

We base our work on an OoO model, inspired by formal models proposed in the literature and illustrated in Fig. 2, to reason about timing modeling [5], [15], [24]. The model is based on a pipeline containing an in-order front-end responsible for fetching (IF) and decoding (ID) the instructions from the input sequence, an in-order back-end so as to commit (COM) the instructions in program order via a reorder buffer (ROB), and an OoO execution engine in the middle, which may hold instructions in Reservation Stations (RS) and perform computations in Functional Units (FU). The model can be parameterized using four parameters: $superscal$ determines the number of instructions fetched/decoded/committed per cycle, N_{FU} specifies the number of RSs/FUs, and N_{RS} and N_{ROB} specify the sizes of the buffers of the RSs and the ROB.

The model allows the execution of an arbitrary specified instruction sequence. We specify for each instruction its data dependencies and the admissible FUs. We also specify for each instruction the sets of possible latencies for the FUs and the IF stage (the COM stage takes 1 cycle). The instruction sequence, the choice of FUs, as well as the choice of latencies explicitly represent the *initial state*. The set of initial states is given by all combinations of these choices.

Starting from an initial state, instructions deterministically advance through the pipeline at each cycle (→). The OoO computation relies on Tomasulo's algorithm [25] and only represents how instructions progress through the pipeline, i.e., the instructions' computations are not modeled. Instructions are issued directly from the ID stage to a FU (⋯→) or otherwise from the associated RS. The results from FUs are bypassed (- -→), allowing the back-to-back execution of dependent instructions on FUs. The RSs and the ROB keep track of the status of instructions (pending/ready/executing/completed/committed) and their (data) dependencies (↔). Instructions are assigned an entry in the relevant RS and in the ROB within the ID stage. If the buffer capacity of one of these resources is reached (N_{RS} or N_{ROB}), the pipeline is stalled in ID (otherwise this stage takes 1 cycle).

Remark — The hardware model is centered on the pipeline stages. In particular, the memory system is modeled implicitly, through variable latencies. Possible interference, e.g., on the memory bus, are thus not modeled. This microarchitecture is prone to TAs, but sufficiently simple in order to reason about the relevant events that may occur during execution. This is also true for the choices of initial states—extensions are discussed in Sec. VII.

In order to reason about TAs, we need to extract information from these execution traces. For this, we define two functions to capture the runtime behavior at the hardware level:

Definition 2: Events — The function *Events*: $\mathcal{S} \rightarrow \mathcal{P}(\mathcal{E})$ (where $\mathcal{P}$ denotes the power set) provides a set of triples $(i, r, t) \in \mathcal{E}$, called timestamped events, where i is an instruction identifier, r a resource identifier, and t a timestamp.

The *Events* function captures from an execution trace (Def. 1) any timestamp t when an executed instruction i triggers an event associated with a resource identifier r, which may refer to an in-order commit unit or the acquisition/the release of a relevant hardware resource by an instruction.

Definition 3: Instruction Dependencies — The function *IDeps*: $\mathcal{S} \rightarrow \mathcal{P}(\mathcal{D})$ provides a set of triples $(i, j, t) \in \mathcal{D}$ where i and j are instruction identifiers and t a timestamp.

The *IDeps* function captures from an execution trace the data or control *dependencies* that impact an event of instruction j at timestamp t due to instruction i.

Instantiation 2: Events & Instruction Dependencies

- In our case, instruction identifiers i are capital letters, e.g., A or B, according to the order of executed instructions.
- Instructions progress through the OoO pipeline; the fetch unit (IF stage), the decode unit (ID) and the FUs are considered to be resources that are acquired when the instruction enters the unit and released when the instruction exits. The acquisition/release of one of these resources is denoted by an arrow pointing up-/downward, followed by the name of the unit/stage, e.g., $r = \uparrow \text{IF}$ or $\downarrow \text{FU}_1$. Instructions complete in the COM stage, which is reflected by a resource $r = \text{COM}$. Each instruction is assigned an entry in the ROB/RS buffers in ID. The attribution of an entry is indicated by $r = \text{ROB}$ and $r = \text{RS}$ respectively.
- In our case, an instruction dependency of j on i may impact j at a single timestamp t, i.e., the timestamp of the acquisition of a FU by i. The *Events* and *IDeps* functions emit events/dependencies according to the progress of

instructions in the pipeline as defined by Inst. 1.

The fetch unit and the FUs are considered relevant resources in our model, since they may have an intrinsic impact on the timing of other events. The time that an instruction spends in these resources thus represents a latency that can be explained directly by the *initial hardware state*. Note that $\uparrow$ / $\downarrow$ IF exclusively correspond to the time required for fetching an instruction (from memory or a bus): additional stalling may occur in the IF stage *after* $\downarrow$IF, e.g., when the instruction in ID stalls. Also, the ID stage is relevant since its timing is not determined only by the use of the fetch unit. The COM stage is relevant since it represents the completion of instructions, therefore a reference point when comparing traces. Since COM events are simple end-markers (they take a single cycle and may never cause stalling), there is no need to distinguish acquisition/release. Similarly, ROB and RS are simple markers within the ID stage (the behavior/content of ROB and RS are otherwise irrelevant for our approach).

Remark — While we focus on our OoO model, we would like to make some remarks relevant for more general architectures:

- We expect that events coincide with register writes in most cases, e.g., when data of an instruction is written into a pipeline register.
- On more complex architectures, it might not be sufficient to capture events only for pipeline stages. Events related to caches, buses, memories, etc. might be required.
- On real processors, it is not sufficient to fix the instruction sequence in order to be certain that the exact same program was executed, e.g., due to changes in the input data. In this case, registers and memories visible through the instruction set architecture (defined by the programmer's manual or application-binary interface) have to be identical, whereas hidden registers and memories may diverge.

B. Event Time-Dependence Graph (ETDG)

The ETDG captures a minimal duration imposed between two events in a trace τ. The nodes are the events in the trace (Def. 2) and the arcs connect two nodes for which the source node *may* have a direct timing impact on the other.

Definition 4: Event Time-Dependence Graph (ETDG) — The ETDG of a trace τ is a graph $G = (\mathcal{N}, \mathcal{A})$, where $\mathcal{N}$ is the set of nodes, which are directly derived from the events occurring during the execution of the trace (i.e., $\mathcal{N} = \mathit{Events}(\tau)$), and $\mathcal{A}$ a set of weighted arcs, which specify *timing dependencies*. The arcs (and their weights) are derived from *microarchitecture-dependent rules* using information on the trace's events as well as on data dependencies among instructions (Def. 3). An arc is denoted as $e_1 \xrightarrow{w} e_2$, where e_1 is the source event node, e_2 the destination node and w the weight, i.e., a minimal delay imposed between the events.

Instantiation 3: ETDG

Our microarchitectural model imposes order and delay constraints between events, captured by the following rules (which may be applied in any order).

1) **Order of pipeline stages**: The pipeline structure imposes a progression order wrt. a given instruction, as well as a minimal duration. In terms of events, any instruction X of a trace has nodes of the form $(X, \uparrow\mathrm{IF}, t_1)$, $(X, \downarrow\mathrm{IF}, t_2)$, $(X, \uparrow\mathrm{ID}, t_3)$, $(X, \downarrow\mathrm{ID}, t_4)$, $(X, \uparrow\mathrm{FU}_i, t_5)$, $(X, \downarrow\mathrm{FU}_i, t_6)$ and (X, COM, t_7). Instructions are decoded in a single cycle, so: $(X, \downarrow\mathrm{IF}, t_2) \xrightarrow{0} (X, \uparrow\mathrm{ID}, t_3), (X, \uparrow\mathrm{ID}, t_3) \xrightarrow{1} (X, \downarrow\mathrm{ID}, t_4), (X, \downarrow\mathrm{ID}, t_4) \xrightarrow{0} (X, \uparrow\mathrm{FU}_i, t_5) \in \mathcal{A}$. Similarly, since an instruction could be committed immediately: $(X, \downarrow\mathrm{FU}_i, t_6) \xrightarrow{0} (X, \mathrm{COM}, t_7) \in \mathcal{A}$.
2) **Resource utilization**: The duration between the acquisition of the IF stage or a FU (by an instruction) and the matching release (by the same instruction), i.e., the duration of a resource use by the instruction, is determined by the *initial hardware state* (cf. Inst. 1). Hence, considering the events evoked in Rule 1, the weights of the related arcs are exactly the timestamp difference between events: $(X, \uparrow\mathrm{IF}, t_1) \xrightarrow{t_2-t_1} (X, \downarrow\mathrm{IF}, t_2)$, $(X, \uparrow\mathrm{FU}_i, t_5) \xrightarrow{t_6-t_5} (X, \downarrow\mathrm{FU}_i, t_6) \in \mathcal{A}$.
3) **Order of instructions in the input sequence**: If we consider two successive instructions in the input sequence, the stages of the in-order front-end and back-end (cf. Inst. 1) cannot process the second instruction before the first one. A minimal duration between these stages is 0: instructions could be processed at the same time (depending on the $\mathit{superscal}$ parameter). Any pair of *successive* instructions X and Y in a trace has events of the form $(X, \uparrow\mathrm{IF}, t_1)$ and $(Y, \uparrow\mathrm{IF}, t'_1)$, which have to respect the program order and thus $(X, \uparrow\mathrm{IF}, t_1) \xrightarrow{0} (Y, \uparrow\mathrm{IF}, t'_1) \in \mathcal{A}$. The same applies for the decode/commit events: for nodes $(X, \uparrow\mathrm{ID}, t_2)$/$(Y, \uparrow\mathrm{ID}, t'_2)$ and (X, COM, t_3)/(Y, COM, t'_3), it follows that $(X, \uparrow\mathrm{ID}, t_2) \xrightarrow{0} (Y, \uparrow\mathrm{ID}, t'_2)$ and $(X, \mathrm{COM}, t_3) \xrightarrow{0} (Y, \mathrm{COM}, t'_3) \in \mathcal{A}$.
4) **Instruction dependencies**: In the OoO engine, the instructions use the resources independently of their order in the input sequence. However, the execution obviously respects data dependencies, which entails an order between *dependent* instructions. Any pair of instructions X and X' of a trace has events of the form $(X, \downarrow\mathrm{FU}_i, t)$ and $(X', \uparrow\mathrm{FU}_j, t')$ (possibly with $i \neq j$) since instructions always require a computation in a FU. If $\mathit{IDeps}(\tau)$ indicates that X' depends on a result produced by X, then: $(X, \downarrow\mathrm{FU}_i, t) \xrightarrow{0} (X', \uparrow\mathrm{FU}_j, t') \in \mathcal{A}$.
5) **Resource contention**
   a) **Execution in a FU**: Even without dependencies, an instruction cannot be issued to its FU if another instruction is already using it. In this case, the instruction is ready but still not executing, it has to wait in the associated RS (cf. Inst. 1). Such competing instructions have events of the form $(X, \downarrow\mathrm{FU}_i, t')$, $(Y, \downarrow\mathrm{ID}, t_1)$ and $(Y, \uparrow\mathrm{FU}_i, t_2)$, with $t_1 < t' \leq t_2$. If two instructions X and Y exhibit such events, then: $(X, \downarrow\mathrm{FU}_i, t') \xrightarrow{0} (Y, \uparrow\mathrm{FU}_i, t_2) \in \mathcal{A}$.
   b) **Limited in-order parallelism**: Instructions may also suffer resource contention in the in-order front-end and back-end, when more than $\mathit{superscal}$ instructions try to access the resources at the same time (cf. Inst. 1). This occurs when two successive instructions are not part of the same fetch bundle or when the second instruction is completed but not committed yet and thus remains in the

ROB (Inst. 1). Any pair of successive instructions X and Y has events of the form $(X, \downarrow\text{IF}, t_1)$ and $(Y, \uparrow\text{IF}, t'_1)$. If $t_1 = t'_1$, then X and Y are not part of the same fetch bundle, s.t. Y is delayed and fetched when the resource is released: $(X, \downarrow\text{IF}, t_1) \xrightarrow{0} (Y, \uparrow\text{IF}, t'_1) \in \mathcal{A}$. Likewise, successive instructions have events (X, COM, t'), $(Y, \downarrow \text{FU}_i, t_1)$ and (Y, COM, t_2). If $t_1 \leq t' < t_2$, then Y is in the ROB and must wait for the end of the ongoing commit: $(X, \text{COM}, t') \xrightarrow{1} (Y, \text{COM}, t_2) \in \mathcal{A}$.

c) **Finite resources**: Stalling occurs in ID whenever the capacity of the finite ROB or appropriate RS is reached (cf. Inst. 1). If the ROB is full, ID is stalled until instructions are removed from the ROB: the new assignment in the ROB occurs *after* the instruction enters ID, and a minimal duration between the commit(s) that immediately precede the new assignment (within ID) and the end of the stalling in ID is 1. If the RS is full, ID is stalled until the end of an instruction's execution in the FU and thus until an entry is freed in the RS: the new assignment in the RS occurs *after* the instruction enters ID, and a minimal duration between this execution end and the end of the stalling in ID is 0. Moreover, in both cases, the next instruction is transitively stalled in IF: a minimal delay of 0 is imposed between the end of the stalling in ID and the acquisition of ID by the next instruction. Any instruction X has events of the form $(X, \uparrow\text{ID}, t_1)$, (X, ROB, t_2), (X, RS, t_3), $(X, \downarrow\text{ID}, t_4)$ and $(X, \uparrow\text{FU}_i, t_5)$, with $t_1 \leq t_2 \leq t_4$ and $t_1 \leq t_3 \leq t_4 \leq t_5$. If $t_1 < t_2$, then stalling occurs due to the ROB and an instruction X' exists with an event $(X', \text{COM}, t_2 - 1)$: $(X', \text{COM}, t_2 - 1) \xrightarrow{1} (X, \downarrow \text{ID}, t_4) \in \mathcal{A}$. If $t_1 < t_3$, then stalling occurs due to the RS and X' exists with an event $(X', \downarrow\text{FU}_i, t_3)$: $(X', \downarrow\text{FU}_i, t_3) \xrightarrow{0} (X, \downarrow\text{ID}, t_4) \in \mathcal{A}$. Let Y be the instruction that follows X. It has an event $(Y, \uparrow\text{ID}, t')$. If $t_1 < t_2$ or $t_1 < t_3$, then Y is transitively stalled: $(X, \downarrow\text{ID}, t_4) \xrightarrow{0} (Y, \uparrow\text{ID}, t') \in \mathcal{A}$.

Remark — A similar reasoning would apply with additional events, resulting for instance from an explicit modeling of the memory system. Note that the number of events with the same timestamp is unlimited and that $\uparrow / \downarrow$ pairs can be nested. Thus, the rules reported above constitute a sound basis for more complex models.

The ETDG captures the utilization of resources by the instructions. This allows us to formally define latencies:

Definition 5: Latency — Given an acquisition event $(i, \uparrow \text{R}, t_\uparrow)$ and a matching release event $(i, \downarrow \text{R}, t_\downarrow)$, the *latency* δ of i wrt. that resource is $\delta = t_\downarrow - t_\uparrow$.

In our case, an arc always exists between these events (cf. Rule 2 of Inst. 3).

C. Relating Events between Traces

Henceforth, we consider two traces α and β that execute precisely the same instruction sequence and for which we want to decide whether a TA exists or not. As such, we need to be able to reason about events that occur in both of those traces and relate events from one trace to events in the other one:

Definition 6: Corresponding Event — The function *CospEvent*: *Events*$(\alpha) \rightarrow$ *Events*(β) maps an event of trace α to its *corresponding event* of trace β.

Instantiation 4: Corresponding Event

For the microarchitectural model from Inst. 1, such a mapping is straightforward. The acquisition/release or occupation related to the IF, ID, and COM stages of a given instruction identifier are simply mapped to the same events of the other trace of the same instruction identifier, i.e., an event $(i, r, t_\alpha) \in$ *Events*(α) is mapped to $(i, r, t_\beta) \in$ *Events*(β). However, instructions may execute on different FUs in the two traces. For events related to FUs we thus simply map to that other FU, i.e., $(i, \uparrow FU_\alpha, t_\alpha) \in$ *Events*(α) is mapped to $(i, \uparrow FU_\beta, t_\beta) \in$ *Events*(β) (similarly for the release of FUs).

Remark — Note that on our hardware model, such a mapping always exists, i.e., for every event in one trace, a corresponding event exists in the other trace. This might not be the case for all models, e.g., when the bus or memory is not accessed due to a cache hit. The *CospEvent* function needs to be adapted accordingly in that case.

Because the corresponding events between the two traces are used to compare latencies (Def. 5), we can define variations that represent a favorable local case:

Definition 7: Variation — Let δ_α be the latency of a given instruction wrt. a given resource (Def. 5) in trace α and δ_β be the latency obtained from the corresponding events (Def. 6). We observe a *variation* if $\delta_\alpha \neq \delta_\beta$, more precisely a *favorable variation* for α (β) when $\delta_\alpha < \delta_\beta$ ($\delta_\beta < \delta_\alpha$).

Similarly we can detect whether an instruction has switched from one functional unit to another:

Definition 8: Resource Switch — A *resource switch* occurs when the corresponding events (Def. 6) of an instruction's resource use in trace α refers to a different resource in trace β, i.e., for $e = (i, r_\alpha, t_\alpha) \in$ *Events*(α) and *CospEvent*$(e) = (i, r_\beta, t_\beta)$ we have $r_\alpha \neq r_\beta$.

In our OoO model, variations arise from the resources IF and FU_i, and resource switches only from FUs.

D. Causality Graph (CG)

Now that we can build the ETDGs of traces α and β, which captures the order as well as timing dependencies among events of the traces, we further refine the graphs in order to capture causality.

Definition 9: Causality — Given an ETDG $G = (\mathcal{N}, \mathcal{A})$, causal arcs form the subset $\mathcal{CA} \subseteq \mathcal{A}$ of arcs $e_1 \xrightarrow{w} e_2 \in \mathcal{A}$ that represent situations where event e_1 has a direct impact on event e_2 in terms of timing. Node $e_1 \in \mathcal{N}$ is causal to node $e_2 \in \mathcal{N}$ iff there exists an arc $e_1 \xrightarrow{w} e_2$ that is causal, i.e., the first event determines the timestamp of the other event.

Generally, not all arcs of an ETDG correspond to this notion of causality, and, consequently, some arcs are removed from the graph, resulting in the causality graph:

Definition 10: Causality Graph (CG) — The CG is the subgraph $C = (\mathcal{N}, \mathcal{CA})$ of the ETDG (Def. 4), where only the arcs that reflect *causality* (cf. Def. 9) of events are retained.

Instantiation 5: CG

For our OoO model (Inst. 1), we distinguish three rules, identifying cases where an arc $e_1 \xrightarrow{w} e_2$ between two events $e_1 = (i_1, r_1, t_1)$ and $e_2 = (i_2, r_2, t_2)$ has to be removed:

1) *Timing gap*: An arc has to be removed when $t_1 + w < t_2$. In this case, another event has to exist that delays e_2 more than the duration w due to e_1, so that e_2's timestamp is not determined by e_1 (at least not via that arc of the ETDG).
2) *Variation*: An arc needs to be removed if it corresponds to a variation (thus e_1 represents a resource acquisition and e_2 the matching release) (Def. 7). Any event e_0 that occurred before e_1 and that is causal wrt. e_1 is no longer causal to any event e_3 that occurs after e_2 (even if e_2 is causal wrt. e_3), since the timestamp of e_3 is not only determined by e_1 but also by the variation that lies between them.
3) *Resource switch*: The same occurs when an instruction switches from one FU to another (Def. 8). The assignment to a FU results from the initial state (cf. Inst. 1), and consequently, this choice also determines the timestamp of later events e_3 according to the scheduling on FUs.

Definition 11: Causal Region — Given a causality graph $C = (\mathcal{N}, \mathcal{CA})$ (Def. 10) and an event $e \in \mathcal{N}$, we define the *causal region* of that event, denoted by $C(e) = (\mathcal{N}_{C(e)} \subseteq \mathcal{N}, \mathcal{CA}_{C(e)} \subseteq \mathcal{CA})$, as the sub-graph obtained from the nodes that are reachable from e.

E. Counter-Intuitive Timing Anomalies

Based on the variations and their causal region, we can now reason about TAs. We formally define them in accordance with the intuitive definition. Nevertheless: 1) The definition is based on a precisely defined variation (Def. 7) in how an instruction uses a *resource*. 2) The causal region (Def. 11) of this variation limits the *scope* of the TA verdict to this region (i.e., not necessarily the whole trace). 3) Contrary to previous definitions, we do not rely on the (absolute) global execution time. Instead, we compare the *relative* time distance of events by using the operator Δ, which computes $\Delta(e_1, e_2) = t_2 - t_1$ for two events $e_1 = (i_1, r_1, t_1)$ and $e_2 = (i_2, r_2, t_2)$.

Definition 12: Counter-Intuitive Timing Anomaly — For $\tau = \alpha$ or $\tau = \beta$, let $e_{\tau\uparrow} = (i, \uparrow r_\tau, t_{\tau\uparrow})$ be an acquisition event and $e_{\tau\downarrow} = (i, \downarrow r_\tau, t_{\tau\downarrow})$ be the matching release event, s.t. $e_{\beta\uparrow} = \mathit{CospEvent}(e_{\alpha\uparrow})$ and $e_{\beta\downarrow} = \mathit{CospEvent}(e_{\alpha\downarrow})$. The event $e_{\alpha\downarrow}$ triggers a *counter-intuitive TA* at an event e wrt. β, iff:

1) **Variation:** α exhibits a *favorable variation* (Def. 7) at $e_{\alpha\downarrow}$, i.e.: $(\delta_\alpha = t_{\alpha\downarrow} - t_{\alpha\uparrow}) < (t_{\beta\downarrow} - t_{\beta\uparrow} = \delta_\beta)$;
2) **Causality:** e is a node of the *causal region* $C(e_{\alpha\downarrow})$ of the variation (Def. 11), i.e., $e \in \mathcal{N}_{C(e_{\alpha\downarrow})}$;
3) **Slowdown:** α exhibits a *relative* slowdown, expressed as: $\Delta(e_{\alpha\downarrow}, e) > \Delta(e_{\beta\downarrow}, \mathit{CospEvent}(e))$.

We can obviously apply the definition with α and β exchanged in order to get TAs for favorable variations in β.

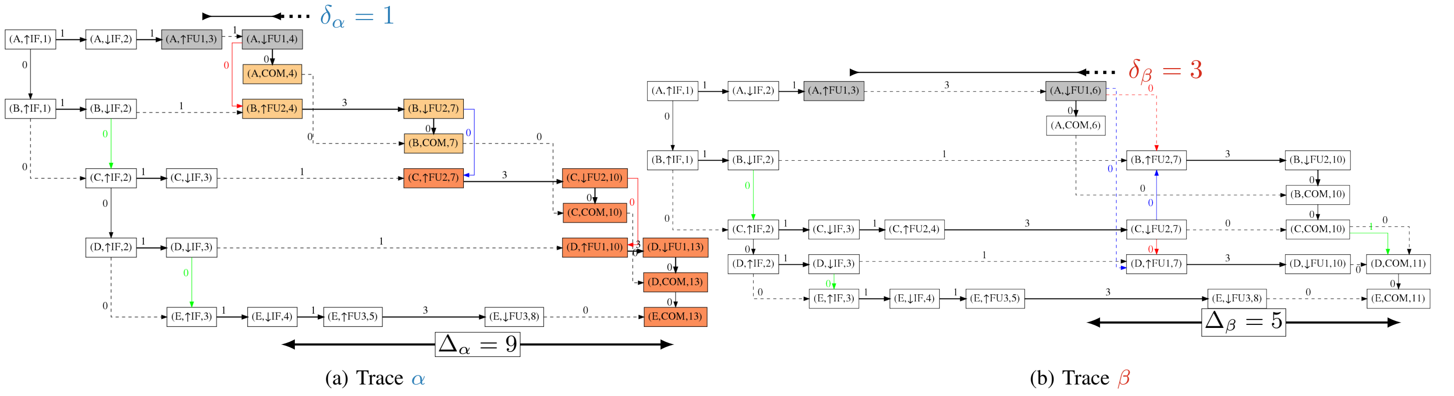
While this definition applies to all events e, we notably focus on COM events. Such events are relevant since the related instructions are fully executed and can no longer impact the execution of other instructions in the trace. However, considering terminal nodes other than COM events, i.e., nodes without any successors in the causal region, is relevant for events representing a resource switch or another variation. This is necessary to reason about the *composition* of variations and about chains of TAs (cf. Sec. VI). In any case, ROB and RS events explain the scheduling but not TAs themselves.

F. Application to the Motivating Example (cf. Sec. III)

Next, we present how the various definitions/instantiations work on the TA pattern of the motivating example (cf. Sec. III), and how our definition addresses the raised issues. As a first step, we consider only traces α and β. Fig. 3 shows the ETDG and CG of this pair of traces—all arcs are in the ETDG while the dashed arcs are not in the CG, which contains only the solid arcs. We illustrate the successive steps of the procedure.

1) The first step consists in extracting *events* from both considered traces of Fig. 1. The derived events (cf. Def. 2/Inst. 2) are the nodes in Fig. 3a and 3b.
2) From these events and from the time-dependence rules of Inst. 3, we build the *ETDG* (Def. 4) of each trace. In Fig. 3a and 3b, the nodes that have the same timestamp are vertically aligned. The arcs derived from Rule 1 (order of stages) and Rule 2 (resource use) are represented with bold black arrows (→), those from Rule 3 (order of instructions) with simple black arrows (→), those from Rule 4 (instruction dependencies) with red arrows (→), those from Rule 5a (contention in a FU) with blue arrows (→) and, finally, those from Rule 5b (limited parallelism) with green arrows (→).
3) Both ETDGs exhibit a single *variation* (Def. 7), namely from the latencies (Def. 5) related to the use of FU_1 by instruction A in both traces, denoted as δ_α and δ_β. The variation is highlighted similarly in Fig. 1 and 3 (■). The variation is favorable for α ($\delta_\alpha = 1 < 3 = \delta_\beta$).
4) Both CGs (Def. 10) are derived from the ETDGs by removing the dashed arrows in Fig. 3, according to Rules 1 and 2 of Inst. 5. Rule 3 does not apply due to the absence of resource switches.
5) We then compute the *causal region* $C(e_{\alpha\downarrow})$ (Def. 11) of the release event $e_{\alpha\downarrow} = (A, \downarrow\text{FU}_1, 4)$ using the CG C of α, which contains the favorable variation. The nodes of this region are highlighted in Fig. 3a (■/■)—reflecting the same information as in Fig. 1.
6) We finally compare the *relative time distance* from $e_{\alpha\downarrow}$ to each event $e \in \mathcal{N}_{C(e_{\alpha\downarrow})}$ of the causal region, with the relative time distance from the corresponding release event $e_{\beta\downarrow} = \mathit{CospEvent}(e_{\alpha\downarrow}) = (A, \downarrow\text{FU}_1, 6)$ to each corresponding event $\mathit{CospEvent}(e)$ in trace β. The events at which a TA manifests (Def. 12) are highlighted with a more pronounced color in Fig. 1 and 3a (■). Let us consider, in particular, the commit event $e_E = (E, \text{COM}, 13) \in \mathcal{N}_{C(e_{\alpha\downarrow})}$. The relative time distance is $\Delta(e_{\alpha\downarrow}, e_E) = 9$, denoted by Δ_α in Fig. 1 and 3a. The corresponding relative time distance in the ETDG of β is $\Delta(e_{\beta\downarrow}, \mathit{CospEvent}(e_E)) = 5$, with $\mathit{CospEvent}(e_E) = (E, \text{COM}, 11)$, denoted by Δ_β.[1] Trace

[1] $\mathit{CospEvent}(e_E)$ may not be in the causal region of the variation in β.



(a) Trace α

(b) Trace β

Fig. 3: ETDG/CG of α and β from the motivating example (cf. Fig. 1). For each trace, all arcs are in the ETDG, while the arcs represented with dashed arrows are not in the CG. The nodes of the causal region of the favorable variation are highlighted (■/■). Among these nodes, those at which a TA manifests are highlighted in a more pronounced manner (■).[2]

α is longer ($\Delta_\alpha > \Delta_\beta$): a TA is triggered by $e_{\alpha\downarrow}$ at e_E.
Let us now bring α' up, so that we consider the three traces that remain after pruning β' from the motivating example. The variation related to the use of IF by instruction E is favorable in α and β against α'. It can be easily observed from Fig. 1 that in both former traces, the causal region of the release event of this variation is limited to the use of FU_3 by E. There is no slowdown in these causal regions wrt. α', and thus no TA is signaled. This correctly reflects the fact that for the considered traces, this variation has no scheduling impact on the other instructions (cf. Sec. III). Besides, the variation in FU_1 is favorable for α' wrt. β. The causal region of the release event in α' is exactly the same as in α, thus entailing the same TAs wrt. β. This consistently captures the TA pattern shared by both traces α and α'.
Finally, our definition does capture the TA pattern even if we consider β' instead of β. In α/α', exactly the same TAs as wrt. β are triggered wrt. β', for two reasons stemming from the fact that our definition is not based on the WCET. We do not exclusively focus on the end of the traces, and moreover we compare relative times.

V. Correctness Arguments

In this section, we show that the detection procedure for TAs on our OoO model, based on the various instantiations from Sec. IV, is accurate wrt. the intuitive understanding of TAs.

A. Prerequisites

We first investigate the correctness of the ETDG, which establishes a link between the actual execution of the input traces on the OoO model (Inst. 1) and our formalization.

Lemma 1: ETDG Accuracy — The ETDG (Inst. 3) is **accurate**, i.e.,
1) its nodes exactly represent the relevant observable events of an execution trace for the study of TAs; 2) for all pairs of events for which an order is imposed by the OoO model during the execution of a trace, arcs exist in the graph; 3) the arc weights represent actual delays imposed by the OoO model between the respective events.

[2]For simplification, ROB and RS nodes are not represented (cf. Sec. IV-E).

Proof 1. *1)* COM *events are indispensable in order to identify executions of the* same *program and to delimit the contribution of a given instruction within an execution of our Transition System* (TS)*. Besides, the acquisition/release of resources (IF/FU) that provide information about the* initial hardware state *must be represented, since the intuitive understanding of TAs relies on latencies ensuing from the initial state. The observation of the* COM *stage (cf. Sec. IV-A) is subject to harmless simplifications, since commit always takes* 1 *cycle. The duration spent in the* RS*s and the* ROB *is an* emerging *property of the pipeline scheduling, not a latency that is directly related to the initial state.*
2) Due to space consideration, we cannot provide a full proof showing that the ETDG reflects any possible evolution of the state variables that represent relevant resources, specified by the transition relation of our TS. However, the microarchitectural model may impose an ordering among events only in four situations (Rules 1/2 and 3-5 of Inst. 1). ■
The intuitive definition of TAs from the introduction furthermore requires that traces are *comparable*.

Lemma 2: Trace Comparison — Inst. 4 as well as Def. 7 and 8, which build on it, allow a **consistent comparison** of the two traces at hand.

The objective of the CG is to capture *causality*, i.e., the relationship of two events within a trace s.t. the first event *explains* why the second event occurred at a specific instant.

Lemma 3: CG Accuracy — The CG (Def. 10) is **accurate**, i.e., 1) all arcs of the graph link causal events; 2) all the events that verify the causality relationship are connected through an arc.

Proof 2. *1) We must ensure that all remaining arcs in the CG correspond to the causality relationship. All arcs are also in the ETDG, so they represent a delay constraint. This constraint is clearly necessary for causality but it is not sufficient. We must verify that the source event* e_1 determines *the timestamp*

of the destination event e_2. Indeed, a timing gap may exist due to the timing dependency of e_2 on another event e_0. Inst. 5 (Rule 1) ensures that such an arc between e_1 and e_2 is not present in the CG. Note that in this case an arc has to exist between e_0 and e_2 due to the second item of the instantiation. Besides, we must verify that the causality relationship takes into account the comparison of both traces at hand, since we are interested in explaining *whether a given event caused a divergence between the two traces. Variations and resource switches are the only source of divergence between the execution of traces (cf. Inst. 1), and they are correctly captured (cf. Lemma 2). All other effects (e.g., the order of computations on* FU*s) are emerging from that in the hardware model. Recall that variations are based on latencies and that the assignments to* FU*s are independent and only depend on the initial hardware state (cf. Inst. 1). Consequently, the instants of the events that are time-dependent on the release e_2 of a resource use that exhibits a variation or a switch are not explained by the matching acquisition e_1 alone. Rules 2 and 3 of Inst. 5 complete the removal of undesired arcs, without loss of relevant information (the ETDG suffices to characterize the resource use in question).*

2) Assume that a pair of events exists that verifies the causality relationship, but no arc in the CG connects them. By definition, the CG is a sub-graph of the ETDG that shares the same event nodes. Clearly, an ordering exists between the two events. Consequently, a path has to exist between the two nodes in the ETDG (due to Lemma 1) and at least one arc along this path was removed during the CG construction. The removed arc either represents a timing gap or a variation/resource switch. In the former case, the initial hypothesis on causality is contradicted. In the latter case, the initial variation is no longer the only explanation for the instant of the destination event. Consequently, an arc must exist between the two considered events. ■

Causal regions represent *chains* of events where each event is time-determined by its predecessor.

Lemma 4: Timing in Causal Regions — Given a causality graph C, for any pair of events (e_1, e_2), where $e_2 \in \mathcal{N}_{C(e_1)}$ (Def. 11), the relative time distance $\Delta(e_1, e_2)$ between the two events **corresponds exactly** to the sum of the arcs weights on any path between the two events of $C(e_1)$.

PROOF 3. *This follows from Lemma 3 and by induction from the fact that any arc of $C(e_1)$ satisfies Rule 1 of Inst. 5.* ■

B. Formal Definition of Timing Anomalies

We now argue that Def. 12 corresponds to the intuitive definition of TAs, considering the notion of causality. Due to lack of space, we will focus on scenarios with a **single variation** and without resource switches. Most existing definitions are limited to this kind of execution scenarios (often without explicitly stating so).

Lemma 5: Counter-Intuitive TAs — For the OoO model from Inst. 1, Def. 12 corresponds to the intuitive understanding of TAs.

PROOF 4. *Given the vagueness, inherent to the* intuitive *understanding wrt. TAs, it is impossible to provide a formal proof. We will thus develop a series of arguments highlighting different aspects of the definition and showing that its verdicts are coherent with this intuitive notion of TAs.*

We first investigate the three necessary conditions at the heart of our definition by assuming the absence of each of them:

1) ***Variation:*** *Suppose that the two input traces do not exhibit any variation. Consequently, the two traces are identical and our definition does not signal a TA—conforming to the intuitive notion of TAs.*

2) ***Causality:*** *Now suppose that a (single) favorable variation is present in one of the traces and that an event e, of that execution trace, experiences a slowdown due to the variation. Furthermore, assume that e is not in the causal region of the release event $e_\downarrow$ of the variation. Our definition does not signal a TA, while intuitively one would expect a TA.*

However, given that the slowdown is due to the variation, some ordering has to exist between $e_\downarrow$ and e, which has to be captured by a corresponding path in the ETDG (Lemma 1). At least one arc along this path was removed according to Lemma 3. The rules of Inst. 5 referring to variations and resource switches are not applicable, since only a single variation occurred. The arc must have been removed due to a timing gap*. This contradicts the hypothesis that e suffered a slowdown due to the variation (e was delayed by some other event) and the verdict of our definition must be correct.*

3) ***Slowdown:*** *Finally, assume that a (single) favorable variation is present in the input traces, that an event e suffered a slowdown due to the variation, and that e is in the causal region of the variation's release event $e_\downarrow$ but e does not exhibit an increase in relative time (Δ) for the favorable trace. Intuitively, a TA should be signaled, due to this* slowdown.

Having a single variation, and no resource switch, means that both traces are identical up to and including the acquisition events of the variation. Given that the acquisition events occur at the same instant, that the variation is favorable (δ), and that the relative time distance (Δ) is not larger in the favorable trace, it follows that also the absolute time of e is smaller in the favorable trace. This contradicts that e suffered a slowdown and the verdict of our definition must be correct.

Clearly, traces that do not satisfy the three conditions in Def. 12 lead to verdict that is coherent with the intuitive notion of TAs. It remains to show that our definition is coherent with this notion when it actually signals a TA.

For this, assume that a (single) favorable variation is present in the input traces and that an event e exists that is both causal wrt. the variation and whose relative time distance increased in the favorable trace, but that did ***not*** *experience a slowdown. In terms of the intuitive notion, no TA should be signaled, while our definition clearly does.*

As before we need to contradict the fact that e did ***not*** *experience an* intuitive *slowdown. For this we can analyze the impact of the relative slowdown on e's timestamp:*

1) If the increase of the relative time distance (Δ) is larger than the amplitude of the variation (δ), its absolute time

becomes larger in the favorable trace. It is difficult to attest the absence of a slowdown for e when both its relative and absolute times increase. This leads to a contradiction and the verdict of our definition must consequently be correct.

2) The increase of the relative time distance (Δ) is not large enough and e occurs at the same time or even earlier in the favorable trace than in the other trace—which leads to a quite controversial situation and the intuitive notion of TAs is no longer sufficient to reach a conclusion.

We argue that our definition still provides a sensible verdict for two reasons. First of all, we can construct examples that reflect the same TA pattern (see Sec. VI) with regard to some event e with the only difference that in one example e occurs earlier and in the other example e occurs later in terms of the absolute time. Since both examples exhibit the same pattern, the verdict should be the same for both examples—which is the case for our definition. Secondly, a strong link between the relative slowdown and causality exists (Lemma 4), which ensures that the accumulated delay up to e in the favorable trace is always greater than that of its corresponding event in the other trace, i.e., the OoO processor performs more work sequentially in the favorable trace between the variation and e. The increase in sequential *work reflects the intuitive notion of TAs even when absolute time is not impacted.* ■

VI. Results from the TA-Detection Procedure

Def. 12 and its application from Sec. IV-F result in a detection procedure. We formalized this procedure using the TLA+ [26] language[3] by instantiating a formal specification of the OoO-hardware model twice (cf. Inst. 1). The TLC model checker thus can explore all possible pairs of traces and automatically identify TAs. Input parameters of the two OoO models specify the common instruction sequence and the data dependencies, as well as the possible execution choices (cf. Inst. 1). This allows us to query the model checker for TAs: does a specific OoO configuration exhibit TAs? Does a given instruction sequence exhibit TAs? May TAs disappear when restraining the initial hardware state?

The subsequent illustrations and examples are all obtained using this tool. The input sequences are based on very short examples found in the literature [5], [6], [13] and adapted in order to highlight interesting features of our definition (e.g., through variations of the parameters of the OoO model, in particular $superscal$ and N_{FU}). Due to space limitations, we use only a compact trace representation similar to Fig. 1.

A. Simple Cases

We start with a series of simple examples by opposing the results obtained using our definition with previous work and the intuitive notion of TAs. We show that these examples. entailing surprising statements about TAs according to the major formal definitions, are correctly handled by ours.

1) Unrelated Variations: Fig. 4 shows two execution traces on an in-order configuration of our OoO model ($superscal = 1$, $N_{\text{FU}} = 1$). The traces contain two variations, where one is favorable for trace α and the other one is favorable for trace β. The other two traces (i.e., resulting from the other two combinations of the variations) might have been pruned. If we try to apply the definition by Reineke et al. [6] to α and β, the situation of the motivating example (cf. Sec. III) is reversed. We can identify a local worst-case path, namely trace β. The variation in FU_1 is a local worst-case for β. Since both traces have already diverged in cycle 5, when the second variation occurs, and the traces consequently do not share the same prefix, no local worst-case is identified for this variation and only the first observed variation serves as a basis to define the local worst-case path. However, the first variation is irrelevant wrt. the global execution time. The scheduling on FU_1 is exactly the same in both traces, and the global execution time depends on the larger latency among both variations, namely the latency of the second one in this case. More generally, while the intuitive definition clearly leads to the absence of TAs, all existing definitions surprisingly state a TA [15].

Our approach splits the favorable trace α into independent parts. The causal region of the first variation in α is limited to the commit event $(A, \text{COM}, 4)$, since the successive instructions do not have data dependencies and they do not experience a resource contention. The relative time distances of the COM event wrt. the variation in both traces is constant (1 cycle). Our definition correctly states the absence of TAs.

[3] We intend to make the code available publicly.

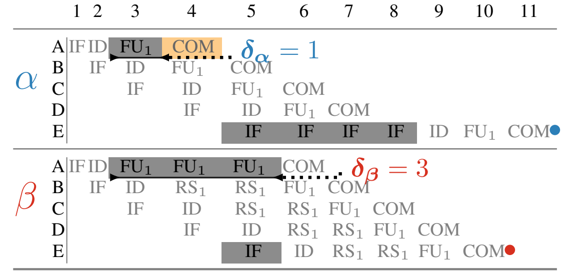


Fig. 4: Illustration of the separation of unrelated variations.

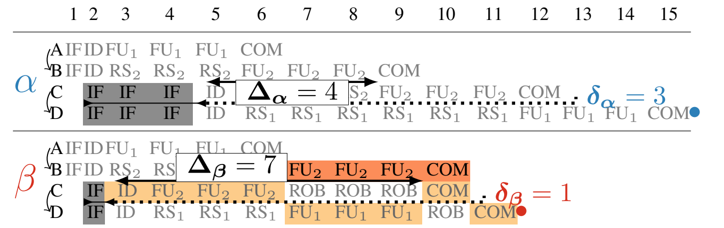


Fig. 5: Example showing that TAs may be limited in scope.

2) TA with a Limited Impact: Since the causal region allows us to precisely capture the scope of a variation, we can also detect TAs that have a limited impact on the execution. In the example from Fig. 5, we clearly observe that the execution of instruction B in FU_2 and its completion in COM occur later in the favorable trace β. However, the trace with the favorable variation has the shorter global execution time. The definition by Reineke et al. [6] does not state a TA, since α is a local worst-case path and is longer. Note that the definition by

Gebhard [8] for instance would signal a TA caused by B [15]. However, it is easy to see that B does not cause the TA.

In our case, the TA is clearly attributed to the variation at instruction C, which blocks instruction B due to a resource contention on FU_2 in Trace β. Such effects are generally not captured in previous work. Our definition also captures the fact that a TA has an effect on a limited scope. In this example, instruction D neither experiences an absolute nor a relative slowdown wrt. the variation. Thus no TA is signaled here.

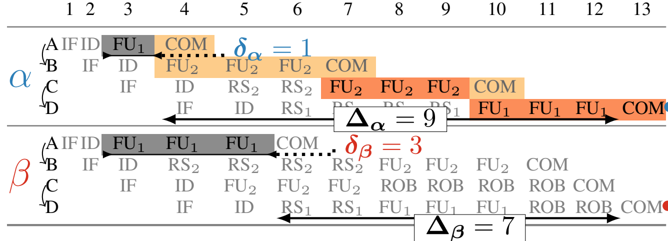


Fig. 6: Example showing a TA pattern that does not impact absolute time.

3) Identification of TA Patterns: Fig. 6 shows a variant of the motivating example, executing on a constrained OoO model ($superscal = 1$, $N_{\text{FU}} = 2$). All the existing definitions signal the absence of TAs [15] due to the identical global execution time. Yet, the global scheduling pattern characterized by the use of FUs is the same as in the motivating example, which clearly exhibits a TA. The situation is similar to the TA detection on traces α/α' vs. β' in the motivating example (cf. Sec. IV-F), but this time the traces exhibit a single variation.

Our definition is based on the precise identification of relevant uses of resources, which leads to the detection of this TA pattern as of the acquisition of FU_2 by C, i.e., the event $(C, \uparrow \text{FU}_2, 7)$. This resource can indeed be used even before the end of the variation in β, i.e., the corresponding relative time distance in β is negative. Moreover, the TA persists up to the end of the execution, although the global execution time is the same in both traces. This may surprise, but our definition relies on the *relative* time distance from the resource release of the variation, in order to capture actual *slowdowns* instead of the absolute execution time. The commit of D in α does suffer a slowdown in α wrt. β due to the sequential execution of instructions B, C, and D.

B. General Scenarios

Next, we consider more complex examples, which exhibit several variations or even FU switches. These considerations are overlooked by all of the existing definitions, though these definitions do not exclude them from their hypotheses. Our approach consistently handles variations by identifying their individual impact through causal regions. Within a causal region, the *relative* time distance enables us to focus on the effects of the last variation, excluding any resource switch. This brings up the problem of the *composition* of multiple variations/switches. If none of the variations taken independently triggers a TA, we might intuitively suspect that the composition of the variations does not exhibit a TA. However, the composition of variations/switches and TAs in general is an open problem. We illustrate this in the following examples.

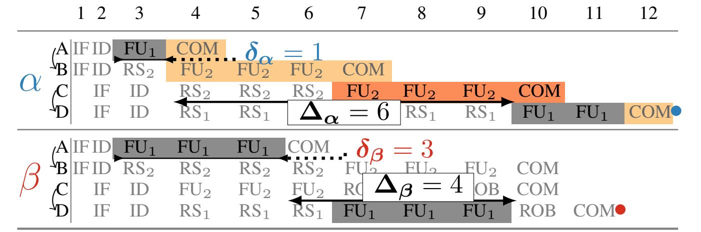


Fig. 7: Composition of two variations with a clear verdict.

1) Serial Composition: In Fig. 7, we consider the slightly modified motivating example, where instruction D also has a variation. Both variations occur one after another in one trace and moreover, the second variation is time-dependent on the first. Differently stated, the causal region of the first variation contains the acquisition event of the second variation. Rules 2 and 3 of Inst. 5 ensure that the causal region ends with this event, similarly to the cases when there is no timing dependency (cf. Sec. VI-A1) or there is a timing gap, since the remainder of the execution behavior does not depend only on the first variation. Consequently, we do detect the same anomaly as previously (cf. Sec. III), triggered by the first variation, up to the commit of C.

Since both variations are time-dependent and favorable for the same trace, this situation leads to a *serial composition*. The second variation only reduces the global execution time in α, it does not entail any particular timing effect nor globally prevent the TA from occurring. We thus could extend the causality region of the first variation in order to compute the relative time distance up to the end of the trace and state a global TA. Similarly, we intuitively suspect that the serial composition of two TAs remains a global TA, since in this case the second variation even exacerbates the already observed slowdown. However, as the subsequent examples show, the analysis of a composition of TAs is complicated in general. Our definition provides a starting point to tackle this problem in future work.

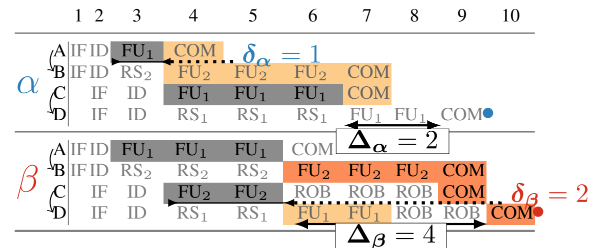


Fig. 8: Illustration of cumulative effects of multiple variations.

2) Composition with a Series of TAs: Let us now consider the example in Fig. 8 in which instructions A and C exhibit variations and instruction C, in addition, switches its FU The definition by Reineke et al. [6] does not signal a TA, since the first variation is favorable for the shorter trace (α).

For our definition, two favorable variations are identified, one for instruction A in α and a second for instruction C in β. The former variation alone does not trigger any TA as indicated

in the figure (■). The latter triggers TAs by itself (■), in particular for D's commit (see Δ_α and Δ_β). This allows us to state that TAs occur in this example.

The characterization of the global timing behavior is tricky due to the interaction between the opposing variations. Let us focus on the favorable variation for α, which clearly does not trigger TAs in α. Note that we observe an increase in the relative time distance wrt. the commit of instruction D (5 cycles in α, vs. 4 in β)—a slowdown. This event is not in the causal region of the variation, since it is exclusively delayed by C. If we focus on Trace β, we observe that the resource switch of C imposes a delay on B due to contention on FU_2. However, the increase of the use of FU_1 by A is crucial in determining the execution order between B and C. The variation on A thus also plays a role in the appearance of the TAs visible in β. Due to the independence of these choices (Inst. 1) causality is excluded though. This shows that the problem of composition needs to be investigated further—notably considering realistic processor implementations where these choices will necessarily expose causal relationships.

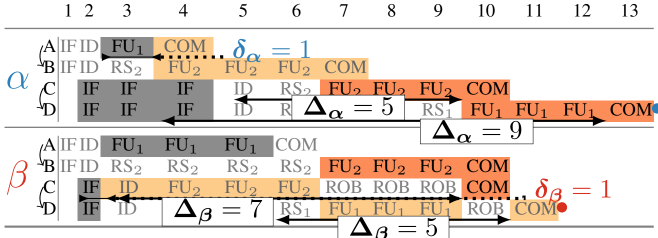


Fig. 9: Composition triggering mutual TAs in both traces.

3) Composition with mutual TAs: Now consider Fig. 9, a combination of the motivating example and the traces from Fig. 5, i.e., a variation in the use of IF by C and D is added. The global execution times and scheduling effects in the resulting traces are the same as in the motivating example: the use of IF by C, though longer, is still too short for its release to delay any relevant event in the trace. The definition by Reineke et al. [6] does not signal a TA, since the first variation is favorable for β, which is also the shorter trace.

Our definition remains consistent, identifying the same TAs for α as in the motivating example. Moreover, TAs are identified due to the favorable variation of IF for C in β, which are consistent with those identified in Sec. VI-A2. The difference is that the commits of B and C occur earlier in α here, which explains the TA for C in β. Our definition thus is able to clearly separate the effects of those mutual TAs.

VII. Conclusion & Future Work

In this work, we proposed a formalization of counter-intuitive TAs based on the notion of causality. This formalization allows accurate reasoning about multiple variations and the resource utilization of instructions through specialized data structures. It also exposes a new problem, that of the composition of timing effects. Finally, a detection procedure is implemented with clear assumptions for an OoO-hardware model, designed to expose exactly the aforementioned features.

In ongoing work, we intend to improve the model with a more concrete scheduler. The actual policy for FU assignments will be relevant when reasoning more deeply about compositions. Compositions will also require additional information linking variations to each other, through side-effects on the hardware state. While short instruction sequences are presented for illustrative purposes in this paper, we also intend to scale up our detection procedure with large software benchmarks.

References

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

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

[3] B. Binder, M. Asavoae, F. Brandner, B. B. Hedia, and M. Jan, "Formal modeling and verification for amplification timing anomalies in the superscalar tricore architecture," *STTT*, 2022.

[4] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm, "Static timing analysis for hard real-time systems," in *Proc. VMCAI*, 2010, pp. 3–22.

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

[6] 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.

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

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

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

[10] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *ISoLA*, 2008, pp. 430–444.

[11] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time system," in *RTSS*, 2002, pp. 279–288.

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

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

[14] 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.

[15] B. Binder, M. Asavoae, B. Ben Hedia, F. Brandner, and M. Jan, "Is this still normal? Putting definitions of timing anomalies to the test," in *RTCSA*, 2021.

[16] J. Reineke and R. Sen, "Sound and efficient wcet analysis in the presence of timing anomalies," in *WCET*, 2009.

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

[18] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. Probst, S. Karlsson, and T. Thorn, "Towards a time-predictable dual-issue microprocessor: The patmos approach," in *PPES*, 2011.

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

[20] A. Gruin, T. Carle, H. Cassé, and C. Rochange, "Speculative execution and timing predictability in an open source RISC-V core," in *RTSS*, 2021.

[21] M. Platzer and P. Puschner, "Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation," in *ECRTS*, 2021.

[22] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for WCET analysis," *Real-Time Syst.*, vol. 34, pp. 195–227, 2006.

[23] Z. Bai, H. Cassé, M. De Michiel, T. Carle, and C. Rochange, "Improving the Performance of WCET Analysis in the Presence of Variable Latencies," in *LCTES*, 2020, pp. 119–130.

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

[25] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of R&D*, vol. 11, no. 1, pp. 25–33, 1967.

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