Revisiting Timing Anomalies in Predictable In-Order Pipelines

Lilia Rouizi
Université Paris-Saclay, CEA List, Palaiseau, France

Mihail Asavoae
Université Paris-Saclay, CEA List, Palaiseau, France

Benjamin Binder
Independent researcher, Paris, France

Lionel Rieg
Grenoble INP - UGA, Université Grenoble Alpes, Verimag, Grenoble, France

Florian Brandner
LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France

Abstract

The quality of timing guarantees ensured through worst-case-execution time analysis and schedulability tests – required to be both sound and precise – is directly influenced by the predictability properties of the execution platform. A platform is considered predictable when *safe* and *precise* bounds can be computed through analysis tools. Counter-intuitive and Amplification *Timing Anomalies* (TAs) are detrimental to predictability and thus may make it much harder/impossible to compute such bounds. In order to address this issue, research has followed two orthogonal approaches, (i) designing predictable execution platforms and (ii) characterizing counter-intuitive TAs through formal definitions. However, predictable designs rarely apply any formal definitions of timing anomalies.

This paper aims at investigating precisely this relationship. We first show how a previously proposed definition of counter-intuitive TAs can be applied to the predictable in-order processor *SIC*. We then extend this approach in order to provide the first formal definition of both counter-intuitive and amplification effects. The proposed definitions are then evaluated on a regular in-order processor as well as the predictable *SIC* core using a systematic approach that allows to assess their applicability and relevance. Finally, we prove, for the first time, the absence of some, but not all, TA effects in *SIC*.

2012 ACM Subject Classification Computer systems organization → Architectures; Computer systems organization → Real-time systems

Keywords and phrases Timing Anomalies, Causality, Timing Predictability, Timing Analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2025.19

1 Introduction

Modern computer architectures strive to achieve better average-case performance using multiple hardware features such as cache memories, branch prediction, intermediate buffers, et cetera. These high-performance processors are often complex and make timing analysis more challenging. This is particularly problematic in safety-critical real-time systems, which require safe and precise timing bounds. To address this issue, *predictable* designs [14,15,28] have been proposed, which typically improve predictability at the expense of average-case performance. Consequently, a compromise between performance and predictability has to be found.

Examining the timing behavior of computer architectures reveals certain timing-related phenomena called timing anomalies (TAs) [23]. Two kinds of TAs exist. *Counter-intuitive* anomalies (CI TAs), manifest as a behavior where a local worst-case scenario (e.g., considering

a cache miss over a cache hit) does not result in the global worst-case timing. The second kind, *amplification* anomalies (AMP TAs), appear as a local slowdown (e.g., due to a cache miss) leading to an even larger increase in the global execution time.

Both kinds of TAs have a significant impact on timing analysis [29]. Firstly, the presence of TAs means that it is unsafe to *prune* local best cases (e.g., cache hits) during timing analysis, since this could lead to unsafe timing bounds. However, considering *all* possible execution scenarios without pruning quickly becomes intractable, even for moderately complex architectures.

Secondly, the presence of TAs can make the result of compositional timing analysis [18] (i.e., combining the analyses of individual architecture components for a complete system analysis) unsound, as analyzing these components individually might not account for certain effects caused by interdependencies and interference between components. Consequently, computing timing bounds by summing up the timing contribution of each component is no longer safe. This is particularly problematic when interactions between different tasks have to be accounted for by schedulability tests, which by themselves can be prone to TA effects [12]. In preemptive systems, additional interference can also emerge because of task switches. If timing bounds are computed in isolation for each task, a preempted task may restart its execution from a state that was not considered in the analysis. This may increase the remaining execution time of that task and potentially invalidate the outcome of the schedulability test. Cache-related preemption delays [4] are an example of such an interference. For instance, a memory access that would normally hit in the L1 cache may instead be forwarded to the L2 cache if the relevant data was evicted due to a preemption. This access creates an indirect interference in the L2 cache and can trigger further evictions, potentially leading to an AMP TA. The presence of TAs may make it impossible to bound the additional execution time caused by that interference [11].

In summary, TAs pose a concrete threat to the soundness of timing analyses and therefore to the safety of critical systems, leading to a growing interest in their definition and characterization.

Several definitions were proposed for CI TAs [5, 7, 11, 23, 27]. However, as detailed in Section 2, these formal definitions sometimes have limitations: they are often not integrated into practical tools, lack applicability at the hardware level or are limited to specific computer architectures, or fail to establish the cause-to-effect link between local and global timings. Furthermore, few papers discuss AMP TAs, even though these amplifications can impact the system's compositionality, which is often a prerequisite to apply timing analyses and schedulability tests.

Since there is no clear-cut definition for these phenomena, the proposed predictable architecture designs [14,15,28] usually do not explicitly refer to any of these formal definitions. Instead, predictability is investigated with regard to a certain analysis model, e.g., static Worst-Case Execution Time (WCET) analysis techniques that prune certain kinds of states. Sometimes the absence of TAs is shown indirectly by proving only conditions that are considered sufficient for the absence of TAs – without explicitly referring to any specific definition. It thus remains open whether the proposed designs are truly free from TAs or whether the predictability property only holds under certain conditions.

This work aims to overcome these limitations by investigating the following key questions:

Q1. How applicable is an existing definition of CI TAs [5] to the predictable processor design *SIC* and the in-order core it is built upon [15]?

Q2. How applicable is this framework to other types of timing anomalies such as AMP TAs?

Q3. Can we systematically evaluate the impact of both kinds of TAs and reason about the actual absence/presence of TAs in the *SIC* processor?

Contributions. We show (Q1) that it is indeed possible to apply the recent definition of Binder et al. [5], which was originally proposed for an Out-of-Order (OoO) pipeline, to in-order cores such as *SIC*. Even more, we provide a complete implementation of a detection procedure in a tool. We then provide (Q2), for the first time, a formal definition of AMP TAs, which can be expressed in the same framework and are thus amenable to the detection procedure. Finally (Q3), we evaluate those definitions on *SIC* [15] using a systematic approach, relying on the aforementioned detection procedures, to investigate the presence/absence of TAs. Our results indicate that *SIC* indeed suffers from similar AMP TAs as the original in-order core it was derived from. However, we provide proofs that CI TAs and certain kinds of AMP TAs cannot occur in *SIC*.

Paper Outline. We first introduce available definitions of timing anomalies and predictable designs in Section 2. In Section 3, we provide a background on the processor models and the CI TA definition used in our work. Section 4 discusses the adaptations for using the chosen CI definition in an in-order pipeline followed by the definition for AMP TAs. The corresponding experiments are presented in Section 5. Section 6 presents formal proofs related to the definitions in the SIC model. Finally, we conclude and present perspectives for future work in Section 7.

2 Related Work

Definitions of Timing Anomalies. The concept of timing anomalies was first introduced by Lundqvist & Stenström [23]. In their paper, they provide three examples that illustrate potential anomalous behaviors. The first anomaly shows that a cache hit may be worse than a cache miss (i.e., a CI effect); the second shows that the miss penalties can be higher than expected (i.e., an AMP effect); finally, the third case highlights an unbounded effect (i.e., called domino effect). Their definition of TAs is based on comparing two execution traces of the same instruction sequence on an OoO processor, and examining the local increase or decrease in timing and its effect on the execution time.

Reineke et al. [27] were the first to formally define CI TAs based on an abstract model of a computer architecture. They compare all execution traces of a program on this model, introducing the concept of locality to represent where the timing differences first become visible between these traces, while also comparing their global execution times.

The definitions of Cassez et al. [7] and Gebhard [11] both follow a similar approach. Essentially, they are each based on the notion of step functions, representing the completion time of instructions. According to the former, a CI TA occurs when the step functions intersect – meaning that a trace initially executing instructions faster becomes slower. The latter compares the step height between execution traces in order to capture the locally favorable execution and then compare the global execution time, i.e., the completion time of the last instruction.

Finally, the most recent definition of CI TAs was formulated by Binder et al. [5]. This definition is the first to establish a relation between local and global timing effects through the concept of *causality*, which ensures that a slowdown actually was caused by the favorable event. It was proposed with a detection procedure for an OoO pipeline.

The first four definitions [7, 11, 23, 27] share a common simplistic way of detecting TAs by only comparing local and global timings between execution traces. Contrary to Binder et al.'s definition, they do not explicitly explain why a local timing change leads to a specific global effect. Finally, their practical applicability remains limited as they have not been implemented and experimented within a tool, and none of them address AMP TAs.

On the other hand, the last definition [5] is specific to an OoO pipeline and the corresponding detection procedure requires building special structures to reason on, which could lead to scalability issues.

We select Binder et al.'s definition [5] to investigate research questions (Q1-Q3) outlined before. This choice is justified as follows: 1) the definition was implemented and tested in a more general modeling framework, covering also other CI TA definitions [6] for an OoO pipeline; 2) this framework comes with excellent tool support; 3) the definition appears to be more reliable and generic than previous attempts [6] and thus appears to be a promising starting point for this work – notably when defining AMP TAs in Section 4.2.2, which are often ignored in the literature.

Timing Predictable Designs. Predictable architectures promise to simplify timing analysis and to increase analysis efficiency. However, challenges such as hardware resource sharing [2] remain significant barriers to achieving predictability. One approach to overcoming these challenges is to implement a form of separation of concerns, which helps to reduce sharing-induced conflicts. For example, several predictable dual-issue pipelines [3, 28] or co-processing units [25] have been proposed, where a separation of concerns occurs at the functional level. More precisely, different pipelines or units handle different types of instructions, i.e, memory accesses vs. regular instructions [3, 28] or scalar vs. vector instructions [25].

Patmos [28] is a Very Long Instruction Word (VLIW) processor with a dual-issue pipeline and a memory system, designed for predictability and including method [9] and stack [1] caches. Patmos maintains the pipeline state unchanged throughout stalling, which, according to the authors, prevents TAs.

A conceptually different approach to developing predictable architectures is advanced by PRET [10], where resource sharing is addressed, among other design features, by a thread-interleaved pipeline to exploit hardware-level parallelism. PRET also incorporates a predictable DRAM controller [26], it favors the use of scratchpad memories over classical caches and is supported by an ISA-level extension with instructions for precise timing.

Even textbook in-order pipelines present timing anomalies due to the shared memory bus and branch prediction [17]. A predictable variant, called SIC [15], is developed based on the principle that a monotonic cycle behavior is sufficient to prevent CI TAs. Accordingly, SIC enforces this behavior, by imposing a predictable memory access and disabling branch prediction. Vicuna [25], a timing-predictable, in-order vector co-processor, adopts the same approach. The co-processor is integrated into a simple in-order RISC-V core [22]. MINOTAuR [14] is another predictable design that follows the same approach, but with the objective of providing higher performance. This design builds on the CVA6 core,[1] and (re-)introduces branch prediction along with parallel instruction execution. The previous architecture designs include properties considered to be sufficient to ensure the absence of TAs and thus, are crafted to ensure predictability. However, none of these designs has been verified against an actual definition of TAs. To address this, we focus on the previously presented SIC core to evaluate the chosen TA definition. This choice is justified by the SIC core's conventional in-order pipeline, which ensures that our approach can be naturally extended to similar architectures. Moreover, SIC is fully implemented, as presented by Reineke et al. in their extended work [16], and is a formally proven predictable processor, making it an ideal reference for validation.

[1] `https://github.com/openhwgroup/cva6`

Formal Verification of Processors. Verification techniques like Model Checking and Deductive Verification are widely used in the field of real time systems, whether verifying the correctness of a microarchitecture or ensuring timing properties.

Model Checking [8] is a formal verification method that systematically explores all possible states of a system to determine whether they satisfy a given property, which can be expressed, for instance, as an invariant. Without aiming for exhaustiveness, notable examples include the work of Jhala et al. [19], which employed compositional model checking to verify the functional correctness of a complete processor microarchitecture, and McMillan [24], who formally verified an implementation of an OoO processing unit based on Tomasulo's algorithm using compositional model checking techniques.

Deductive Verification checks whether a processor satisfies given properties using expressive logics. For this, theorem provers are commonly used to model these processors as programs and formally verify properties about them. Kröning [20] applied this approach to the formal verification of pipelined microprocessors, proving data consistency and liveness using the PVS theorem prover. Similarly, Gruin et al. [13] utilized the Coq Proof Assistant to model the MINOTAuR pipeline and formally prove its monotonic cycle behavior.

In our approach, we use model checking to verify timing properties related to the presence/absence of timing anomalies, as will be detailed later.

3 Background

In the remainder of this work we will apply and extend the CI TA definition of Binder et. al [5] to the in-order pipeline of the *SIC* processor [15]. We will thus provide additional details on these two approaches in the following subsections.

3.1 Processor Models

We consider two variants of an in-order pipeline referred to as *SIC* and *SIC* $^-$. *SIC* is the predictable variant of the pipeline with specific extensions/restrictions that are intended to improve predictability. Most notably, memory accesses for instruction fetches and data accesses follow a specific arbitration scheme [15].

We first discuss in detail *SIC* $^-$, a standard in-order pipeline, which lacks the specific additions for predictability. It consists of 6 stages: IF, ID, EX, MEM, and WB with their usual semantics, and an additional stage ST, where stores are performed asynchronously to the rest of the pipeline as represented in Figure 1. We define S={IF, ID, EX, MEM} and S'={WB, ST}.

Store instructions advance through the pipeline stages of Figure 1 from IF to ST, while all other instructions proceed from IF to WB. Branch prediction is disabled such that no new fetch occurs until the branch's outcome is determined at the EX stage.

All stages, except WB and ST, can potentially experience stalling, i.e., temporarily delaying the execution of an instruction due to data dependencies, resource conflicts, or unresolved branches. The IF, MEM, and ST stages share the memory bus which is a shared memory interface that handles L1 cache misses. In *SIC* $^-$, the bus is accessed on a

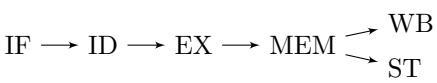


Figure 1 In-order pipeline of SIC [15].

first-come-first-served basis. If two requests are made simultaneously, priority is given to requests from the MEM stage over those from the IF stage. If the bus is in use, any new request will be delayed until it becomes available again. Stores start accessing the bus in the MEM stage for one cycle and maintain access throughout the ST stage.

The different pipeline stages $s \in S \cup S'$ have the following timing characteristics (lat_s):

- The IF stage lasts for one cycle if there is an Instruction-cache (I-cache) hit or for multiple cycles determined by the miss latency (*missLat*) in case of a miss.
- The ID stage lasts for one cycle.
- The EX stage lasts for a number of cycles determined by the instruction's latency.
- The MEM stage lasts for one cycle for all instructions (including stores) except loads. In the case of a Data-cache (D-cache) hit, loads take one cycle, whereas misses take multiple cycles (*missLat*).
- The WB stage lasts for one cycle.
- The ST stage lasts for one cycle if the store hits in the D-cache, or for $missLat - 1$ cycles if it is a miss.

The pipeline is formally described by a mathematical model where each instruction i is associated with a pipeline state $(stage(i), cnt(i))$, which includes the current stage and a counter indicating the remaining cycles in that stage. This counter is initialized to the stage's required latency (cf. lat_s) and decreases until it reaches zero. If a pipeline stall occurs, the counter remains at zero until the instruction can progress to the next stage.

In the predictable variant, *SIC*, instructions follow the same in-order pipeline path as in Figure 1. Additional artificial *pre* and *post* stages represent whether an instruction has not yet entered or has already exited the pipeline. These stages are not part of the pipeline and serve only for the proofs. Instruction progress and branch handling are similar to SIC^- (see above), only the bus access must adhere to a specific arbitration scheme, as discussed next.

The predicate *brpending* is used to constrain branch instructions. It ensures that an instruction i is not allowed to enter the pipeline as long as there is an unresolved branch (i.e., branch prediction is disabled). Additionally, the predicate *mempending* ensures that as long as a data-memory-accessing instruction is pending in the pipeline (hit/miss in D-cache), no access caused by an I-cache miss is allowed to start. In other words, the *memory bus arbiter* ensures that all potential bus accesses are performed in program order.

The previously described features ensure that *SIC* is predictable [15]. In particular, it is shown that the behavior of *SIC* is monotonic, leading to the conclusion that no CI TAs may occur. Monotonicity is defined through pipeline progress, i.e., if a pipeline state c has no more progress than another state c', a monotonic cycle behavior imposes that on the next clock cycle, the evolution of c will have no more progress than the evolution of c'.

		1	2	3	4	5	6	7	8	9	10	11	12	13
SIC^-	A	IF	ID	EX	ex	MEM	MEM	MEM	WB					
	B		IF	IF	IF	ID	EX	ex	MEM	WB●				
SIC	A	IF	ID	EX	MEM	MEM	MEM	WB						
	B		-	-	-	-	-	IF	IF	IF	ID	EX	MEM	WB●

Figure 2 Effect of the bus arbiter in *SIC* compared to SIC^-. Stages using the bus are highlighted in green (MEM/IF). Stalls (idle occupation of a stage) are represented with lowercase stages (e.g., ex). End of traces are represented with bullets (•/•).

Figure 2 illustrates the change introduced by the *bus arbiter* used in *SIC* compared to *SIC* $^-$. In *SIC* $^-$, instruction B requires the bus in the IF stage first, which delays instruction A's request at the MEM stage in cycle 4 and causes the idle occupation of the EX stage by A for one cycle without effective use. On the other hand, in *SIC*, instruction A acquires the bus first since it is an older instruction, and thus delays B's request.

3.2 Causality-based Definition of CI TAs

The definition of CI TAs proposed by Binder et al. [5] is based on a representation of the timing semantics of an OoO pipeline. This representation captures events that illustrate the acquisition and release of pipeline resources (e.g., pipeline stages), as well as the timing dependencies that connect these events.

The OoO pipeline features an in-order front end that fetches (IF) and decodes (ID) instructions. It includes an OoO execution engine that can hold instructions in Reservation Stations (RSs) and perform computations in Functional Units (FUs). Finally, an in-order back end (COM) is responsible for committing instructions in program order using a reorder buffer (ROB).

To formalize CI TAs, the authors introduce the *Event Time-Dependence Graph* (ETDG), a graph where nodes represent the acquisition and release events of hardware resources, and weighted arcs timing dependencies between them. These arcs are created following micro-architecture-specific rules established for the OoO model:

R1. *Order of pipeline stages* to represent the order imposed by the pipeline structure.
R2. *Resource utilization* to express the duration between the acquisition and release of resources.
R3. *Order of instructions in the input sequence* to determine the processing order of a pair of successive instructions.
R4. *Instruction dependencies* to highlight data dependencies between instructions.
R5. *Resource contention* to represent conflicts in accessing resources between instructions.

We explore these rules in greater detail in a comparative analysis presented in Section 4, where we adapt the definition for the in-order pipelines *SIC* and *SIC* $^-$.

The construction of the ETDG is followed by the construction of the *Causality Graph* (CG), a sub-graph of the ETDG, where only arcs that reflect causality of events are retained. Causal arcs are timing dependencies between events such that the first event has a direct impact on the second one in terms of timing. Specifically, the *causal region* of a given node e, is defined as the sub-graph of the CG containing all nodes reachable from e and is essential in defining CI TAs.

The definition of counter-intuitive TAs involves comparing two execution traces, denoted α and β, of the same input program and data on the OoO pipeline. A CI TA is identified if, and only if, the following three conditions are satisfied:

a. **Variation** (δ): a different duration in the use of a pipeline resource by the two traces $(\delta_\alpha, \delta_\beta)$ is observed, either favorable (e.g., $\delta_\alpha < \delta_\beta$) or unfavorable (e.g., $\delta_\alpha > \delta_\beta$).
b. **Relative slowdown** (Δ): a different relative distance from the end of the variation to a given event e is observed in the two traces $(\Delta_\alpha, \Delta_\beta)$, such that there is a slowdown in the favorable trace (e.g., $\Delta_\alpha > \Delta_\beta$).
c. **Causal link**: event e, which experienced the slowdown, is part of the causal region of the variation in the favorable trace.

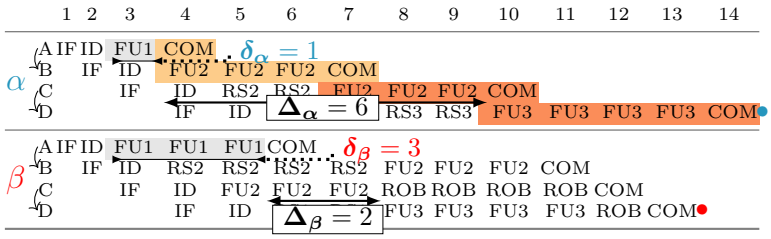


Figure 3 Example of a CI TA detected on the OoO model with the causality-based definition. The arcs (↯) between instructions A and B as well as C and D represent data dependencies. The causal region is represented in varying shades of orange (/), with the darker shade representing events within this region where a TA is detected.

Figure 3 illustrates the definition through a pair of execution traces α and β of instructions A through D, processing the same input data, but starting from distinct initial hardware states.[2] In this example, there are three FUs on which instructions are executed based on their type. The arcs (↯) between instructions A and B as well as C and D represent data dependencies.

a. These traces exhibit a local **variation** (in grey) between the acquisition and release events of FU1 by instruction A, which could represent a data cache hit (δ_α) vs. cache miss (δ_β).
b. The **relative slowdown** is characterized in trace α for instance at the release event of FU2 by instruction C. The relative distance as represented in the figure is $\Delta_\alpha = 6$ cycles in trace α compared to $\Delta_\beta = 2$ in trace β.
c. Finally, the **causal region** of the variation (in orange) starts from the end of the variation in trace α. The first causal link is established through the dependency between A and B, the second one is due to the contention on FU2 between instructions B and C.

Consequently, the definition identifies a CI TA in this example. [3]

4 TA Definitions in In-order Pipelines

The causality-based definition for CI TAs [5] was originally implemented for an OoO pipeline. We now investigate its applicability to an in-order pipeline, which first requires constructing the corresponding ETDG for an in-order pipeline. This ETDG contains nodes representing resource acquisition and release events and arcs built following similar rules as for the OoO pipeline. We expand the set of resources to also include the bus, in addition to the pipeline stages. However, bus accesses are not explicitly represented as dedicated ETDG nodes – since the bus usage coincides with the IF, MEM, or ST stages, respectively. In the coming illustrations the bus usage is similarly indicated by highlighting the stages in green.

In addition, applying the ETDG rules requires architecture-specific adaptations to ensure compatibility with an in-order pipeline.

One such example concerns the pipeline stalling since it is necessary to distinguish between a pipeline resource's **idle occupation** and its **effective use**. Idle occupation is an effect of stalling, an instruction that cannot advance to subsequent stages is stalled in its current stage and thus occupies it without use. On the other hand, effective use is identified as an instruction that uses a stage for one or multiple cycles for actual computations. This

[2] We do not provide an ETDG in this section, as one is already presented in [5] (Figure 3). An example based on our definition is provided in Section 4.
[3] This provides an overview of the content in [5]; for further details, please refer to the paper.

distinction is important to identify variations, which refer to changes in the duration of a resource's **effective use**, as well as to define causality. For brevity, we refer to *SIC* and *SIC* $^-$ in what follows as *in-order models*.

4.1 ETDG Adaptations

Table 1 summarizes the different adaptations of the ETDG rules and emphasizes the relevant differences between the in-order and the OoO pipelines. We use three colors to indicate the extent of the implemented changes: ■ for major, ■ for moderate, and □ for minor changes.

Table 1 Adaptation of the ETDG rules to the in-order architecture with the extent of the implemented changes (■ for major, ■ for moderate, and □ for minor changes).

ETDG rule	In-order pipeline	Comparative aspects
R1. Order of pipeline stages ($\rightarrow$)	Any instruction X that goes into the in-order pipeline has nodes of the form: $(X, \uparrow s, t_{s1})$, $(X, \downarrow s, t_{s2})$, $s \in S \cup \{WB\} \mid s \in S \cup \{ST\}$ (in the specified order). t_{s1}: acquisition timing of s, t_{s2}: release timing of s.	Parallel stages ST and WB are not subject to this order rule and operate independently (in parallel) of each other.
R2. Resource utilization ($\rightarrow$)	We distinguish the idle occupation time of a pipeline resource s by an instruction X from its effective use. In this rule, we represent the effective use of s through the latency lat_s. From R1 & R2 emerge arcs of the form: $(X, \uparrow s, t_{s1}) \xrightarrow{lat_s} (X, \downarrow s, t_{s2})$	The presence of **stalls** in the in-order pipeline introduces a distinction between a resource's effective use and its idle occupation.
R3. Program Order ($\rightarrow$)	Instructions execute in program order, i.e., for two successive instructions X and X', the release of a resource by the preceding instruction X allows the subsequent instruction X' to acquire it. Except for $s \in S'$ if one of X or X' is a store and the other is not. If $t_{s2} = t_{s1}$, then instruction X did not experience any stalling in s, which means that the stage was fully used. $(X, \downarrow s, t_{s2}) \xrightarrow{0} (X', \uparrow s, t_{s1})$	Successive instructions requiring WB and ST allow these stages to be used in parallel, which might not respect the order of instructions.
R4. Instruction Dependencies ($\rightarrow$)	An instruction X can have a data dependency with a previous instruction X'. If so, the corresponding arc connects the release of $s \in \{EX, MEM\}$ by X' (i.e. the data is ready) and the acquisition of EX by X (i.e. it can start the execution): $(X', \downarrow s, t_{s2}) \xrightarrow{0} (X, \uparrow EX, t_{EX1})$	Dependencies emerge through data forwarding from previous EX/MEM stages and the EX stage of the dependent instruction.
R5. Contention **R5.1** In EX/MEM ($\rightarrow$) **R5.2** Bus ($\rightarrow$)	A contention on a resource $s \in \{EX, MEM, ST\}$ occurs when instruction X occupies the EX or MEM stage for multiple cycles, causing a delay in the use of these stages by a subsequent instruction X'. This can be due to a multicycle latency or a D-cache miss. Producing arcs of the form: $(X, \downarrow s, t_{s2}) \xrightarrow{0} (X', \uparrow s, t_{s1})$ A bus contention involves resources s and $s' \in \{IF, MEM, ST\}$ and occurs when one stage delays another by occupying the bus, leading to the following arcs: $(X, \downarrow s, t_{s2}) \xrightarrow{0} (X', \uparrow s', t_{s'1})$	Resources that are concerned with possible contentions are: EX and MEM stages, and the bus. A new rule is added to model the bus resource in the in-order pipeline. Subrules on limited parallelism and finite resources do not apply to the in-order architecture.
R6. *mempending* ($\rightarrow$)	An instruction X that experiences an I-cache miss, has to wait for all previous memory accesses to be performed before the instruction is fetched. We call X' the load or store instruction that has the last memory access before X, with $s \in \{MEM, ST\}$: $(X', \downarrow s, t_{s2}) \xrightarrow{0} (X, \uparrow IF, t_{IF1})$	This rule applies specifically to *SIC* because of the properties enforced upon it.

R1. Minor changes are needed for this rule since both in-order and OoO pipelines have similar structural orderings in terms of pipeline stages. In particular, the parallel stages ST and WB are not subject to this order rule as they operate independently of each other.

R2. Contrary to the OoO model, where instructions in reservation stations and reorder buffer were not explicitly represented through nodes in the ETDG, the **idle occupation** and the **effective use** of a resource need to be distinguished in in-order pipelines. In both *SIC* variants, instructions immediately *use* the resources of a stage upon entering it. However, as previously detailed, an instruction might idly occupy a stage longer than its effective use. We introduce new ETDG arcs to express this distinction.

R3. Despite allowing OoO computations, the OoO pipeline ensures that instructions are committed in order. However, in the considered in-order pipelines, instructions may complete in a different order than the program order. This is due to the parallelism between the WB and ST stages, e.g., more recent arithmetic instructions may complete while a store experiences a miss in the ST stage.

R4. This rule remains the same, but it is important to highlight the change in the structural elements of the dependency. In the OoO pipeline, these dependencies appear between different FUs. In the in-order pipeline, an instruction's result may become available in the EX (regular instructions) or MEM (load instructions) stages respectively. Dependencies thus appear between the EX/MEM stages and EX.

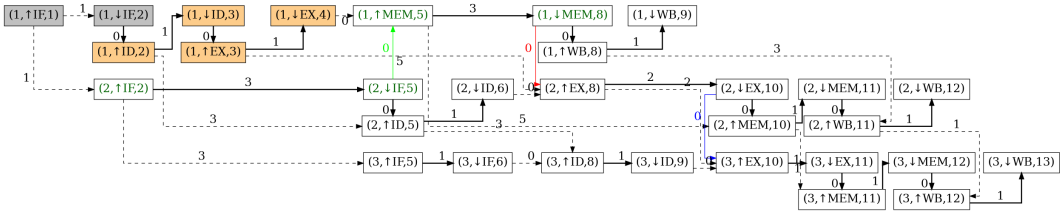
R5. This rule is divided into two subrules. The first subrule (R5.1) is similar to the OoO pipeline with few structural changes (similar to R4). However, the bus contention subrule (R5.2) is new and did not exist in the OoO model. Contention on the bus can occur between the IF, MEM, and ST stages that need to access the main memory in case of a cache miss. On the other hand, subrules about *limited parallelism* (possible parallel execution in case of a superscalar pipeline) and *finite resources* (effect of limited size of reservation stations and reorder buffer) do not apply in the in-order pipeline.

R6. The operation of the predictable *SIC* core also introduces a new timing dependency that did not exist in the original OoO model. We thus introduce new arcs representing *SIC* specific behavior, e.g., due to the *mempending* predicate.

As before, the causality graph is derived from the ETDG by keeping only arcs that represent timing dependencies which directly impact the timing between two events. These timing dependencies can be represented by any of the previously defined arcs R1-R5, and R6 for *SIC*.

In Figure 4, we present the constructed ETDG for trace α shown in Figure 4b for SIC^-. The bus resource is visible in the MEM nodes of instruction 1 and the IF nodes of instruction 2. This example is exhaustive and contains all types of arcs, except R6. The arc between the acquisition and the release of the ID stage by instruction 1 is an example of R1 and R2. R3 appears, for instance, in the arc connecting the release of the ID stage at cycle 3 by instruction 1 and the acquisition of the EX stage. At cycle 5, the green arc between the IF stage of instruction 2 and the MEM stage of instruction 1 represents subrule R5.2 (both instructions suffer a cache miss, causing bus contention, cf. Figure 4b). The red arc at cycle 8 shows the dependency R4 between instructions 1 and 2. Finally, at cycle 10, the blue arc indicates a resource contention in the EX stage R5.1 between instructions 2 and 3.

The ETDG in the figure also contains dashed lines, representing arcs where causality is broken. We consider *stalls as causality breakers*, since a stalled instruction occupies the stage without using it, and thus does not block subsequent instructions from advancing due to its own use. For example, in Figure 4 causality breaks for instruction 1 at the EX stage due to stalling. This highlights the important distinction between resource idle occupation and effective use.



(a) ETDG constructed for the execution trace shown below.

	1	2	3	4	5	6	7	8	9	10	11	
α ↓1	IF	ID	EX	ex	MEM	MEM	MEM	WB				
↓2		IF	IF	IF	ID	id	id	EX	EX	MEM	WB	
3					IF	if	if	ID	id	EX	MEM	WB●

(b) Execution trace corresponding to the ETDG from above.

Figure 4 ETDG (a) for an execution trace (b) executing on the in-order pipeline SIC^-. Nodes (X, s, t) represent events of instruction X acquiring (↑) or releasing (↓) a resource s at time instant t. Arcs represent timing dependencies between events.

4.2 CI and AMP TA Definitions

Based on the construction rules for the ETDG, Causality Graph (CG), and causal regions, we have the necessary tools to define TAs by extending the original work by Binder et al. [5]. One limitation of this approach reasons about causal regions originating at a variation. Consequently, only effects that manifest *after* the variation can be considered. AMP TAs may sometimes delay the variation itself, it is thus necessary to reason about events that occur before it. We call the impact of such events *pre-effects*, which will appear in our definitions as Δ_{Pre}.

In order to define the delays imposed by *pre-effects*, we consider $v = (X, \uparrow s, t_{s1})$, the acquisition event of the variation, which in our case may refer to a stage s. We first identify all events $blk_r = (\_, \uparrow s', t_{s'1})$ that may block the acquisition of a resource r out of a set SR_s needed at s. In our case SR_s can either be the bus or, in the case of *SIC*, a virtual resource *mempending* (necessary due to blocking effects cf. Rule 6). Since blk_r blocks v, the causal region of blk_r has to cover v. We then compute the relative time distance between these events as $\Delta_{blk_r} = t_{s1} - t_{s'1}$. Taking the maximum over all events and resources, represents an upper bound Δ_r of the blocking time due to r before v can occur. Next, we calculate the real waiting time for v by examining the moment when v could have occurred through the ETDG but was delayed due to the blocked resource. This is represented by $\Delta_{wt_r} = t_{s1} - t_{ID1}$ the timing of event $(X_p, \uparrow ID, t_{ID_1})$ in case of a variation in IF (i.e., $s = IF$), where X_p represents the previous instruction to X. On the other hand, if $s \in \{MEM, ST\}$, this timing is $\Delta_{wt_r} = t_{s1} - t_{EX2}$ with $(X, \uparrow EX, t_{EX_2})$. Finally, Δ_{Pre} is defined as the maximum over all resources in SR_s, given by $\Delta_{Pre} = max_r \in {}_{SR_s}(min(\Delta_r, \Delta_{wt_r}))$.

On the other hand, Δ_{Post} is the same relative slowdown computed in the causality-based definition for OoO pipelines. It covers effects that come after the variation (post-effects).

4.2.1 Definition of CI TAs

For CI TAs, we merely instantiate the definition from Binder et al. for the in-order models, using the same three conditions that were established for the OoO pipeline. However, due to the bus modeling, we now capture the impacts of the variation both before and after

the variation itself. The prior impact induced by the variation primarily result from bus unavailability, whether due to the bus being in use or because access is blocked by specific arbitration mechanisms. As a result, this prior impact can only be observed in the trace with the cache miss. Nevertheless, for consistency, we apply it to both traces.

▶ **Definition 1.** *Considering two execution traces α and β of the same instruction sequence and input data on the in-order models, trace α is said to exhibit a CI TA if, and only if:*

a. ***Variation**: α exhibits a favorable variation in terms of resource usage, i.e., $\delta_\alpha = lat_{s\alpha} < \delta_\beta = lat_{s\beta}$ (lat_s: latency of stage s – cf. Section 3).*
b. ***Relative slowdown**: α exhibits a relative slowdown for an event e, that can appear prior (pre) or following (post) the variation, i.e., $\Delta_\alpha = \Delta_{Pre\alpha} + \Delta_{Post\alpha} > \Delta_\beta = \Delta_{Pre\beta} + \Delta_{Post\beta}$, where Δ_{Pre} and Δ_{Post} represent, respectively, the prior and post impacts of the variation in both traces α and β.*
c. ***Causal link**: event e, experiencing the slowdown, is part of the causal region of the observed variation in trace α.*

4.2.2 Definition of AMP TAs

In the following we will propose, the first, formal definition of AMP TAs, which is applicable to the in-order models.[4] To accomplish this, we adopt a similar approach to that used for defining CI TAs, resulting in a new definition for the amplification effect.

▶ **Definition 2.** *Considering two execution traces α and β of the same instruction sequence and input data on the in-order models, trace α is said to exhibit an AMP TA if, and only if:*

a. ***Variation**: α exhibits an unfavorable variation in terms of resource usage, i.e., $\delta_\alpha = lat_{s\alpha} > \delta_\beta = lat_{s\beta}$.*
b. ***Relative slowdown**: α exhibits a relative slowdown at a given event e, i.e., $\Delta_\alpha = \Delta_{Pre\alpha} + \Delta_{Post\alpha} > \Delta_\beta = \Delta_{Pre\beta} + \Delta_{Post\beta}$, where Δ_{Pre} and Δ_{Post} represent, respectively, the prior and post impacts of the variation in both traces α and β.*
c. ***Causal link**: event e, which experienced the slowdown, is part of the causal region of the observed variation in trace α.*

▶ Remark. In our study, the prior impact can only be caused by delayed bus accesses. Since a cache hit does not use the bus, it can only appear in the trace exhibiting a cache miss. This entails $\Delta_{Pre\alpha} = 0$ in Definition 1 and $\Delta_{Pre\beta} = 0$ in Definition 2.

5 Experiments on the In-order Models

We formalize the in-order pipelines *SIC* and *SIC*$^-$ in the TLA+ language [21], opting for a cycle-accurate state update and several parameters such as variations on cache hits/misses and values for cache miss latencies. We formalize the CI and AMP definitions as predicates in TLA+, which, in combination with the TLC model checker, give us executable detection procedures for both kinds of TAs. [5] These procedures can be applied to pairs of execution traces, along with additional input constraints, such as the desired form/location of variations, instruction latencies, et cetera, and yields a verdict: either a TA is detected, or not.

[4] We expect that this definition, like the definition of CI TAs, is also applicable to other models, such as Binder et al.'s OoO model.

[5] We used open-source code from [5] available in `https://bitbucket.org/benjaminbinder/ta-models/src/detection_procedure/`

In our experiments we aim to expose interesting cases of TAs using an exhaustive exploration of traces emerging from programs with a bounded number of instructions. For this we develop an automated test generation procedure that: 1) produces all possible input *scenarios* over all input programs up to a given length, covering specified variations of hits/misses for the D- and I-cache; 2) invokes the aforementioned detection procedures on all pairs of traces that emerge from the input scenarios by launching the TLC model checker on them, and finally, 3) composes a collection of those pairs of traces exhibiting any of the two kinds of TAs.

We define the following test configuration for the experiments. Firstly, we consider programs consisting of a sequence of arithmetic, load, and store instructions, with latencies (lat_s) of 2, 1, and 2 cycles, respectively. Branches are not considered. Instructions that suffer a cache miss are assumed to access the bus for 3 cycles. The program length is limited to 4 instructions for SIC^- and 5 for SIC. Secondly, we consider a *single variation* and analyze two cases: one with a variation in the IF stage and another one with a variation in the MEM stage of a chosen instruction (second or third). Finally, we adopt a write-back policy for store instructions, meaning that on a cache hit, a store instruction accesses only the cache, while on a miss, it accesses the main memory (we exclude additional write-backs that may be required).

For 4-instruction programs, we generate approximately 1,000 different scenarios, resulting in over 450,000 pairs of traces. For the 5-instruction programs, we generate around 6,000 scenarios, yielding more than 21 million pairs of traces. The experiments are conducted on a machine with an 11th Gen Intel(R) Core(TM) i7-1185G7 processor featuring 4 cores and 8 threads, 32 GB of RAM, running Ubuntu 24.04.1, and using TLC version 2.19.

The tests on 4-instruction programs took 16 hours to complete, while the tests on 5-instruction programs took 8 days.

During our experiments, we did not detect CI TAs for either SIC or SIC^-. A proof for the absence of CI TAs in SIC is presented in Section 6. Our experiments did, however, identify AMP TAs for both SIC and SIC^-. We next present some relevant examples of AMP TAs detected by our procedure.

5.1 SIC Motivating Example

We first revisit the motivating example provided in the original SIC paper [15] – which is of course detected by our procedure. In SIC^-, this anomaly is triggered due to the order of bus accesses and is represented in Figure 5. Instructions B and C encounter a D-cache miss, and instruction D an I-cache miss, in both traces. In trace β, instruction A results in a D-cache hit, allowing instruction D to use the bus during the IF stage, which causes a delay in B's access (C as well). Conversely, in trace α, instruction A experiences a D-cache miss preventing D from accessing the bus. When the bus becomes available, priority is given to B and C afterward, thereby amplifying the delay experienced by D.

Applying the conditions from Definition 2, this example is indeed classified as an amplification TA, as follows:

- **a.**✓ There is a local **variation** in the MEM stage for instruction A, satisfying $\delta_\alpha = 3 > \delta_\beta = 1$.
- **b.**✓ We observe a **relative slowdown** in trace α for all stages of instruction D. For example, the release event of the WB stage satisfies $\Delta_\alpha = \Delta_{Post\alpha} = 12 > \Delta_\beta = \Delta_{Post\beta} = 9$. Note that the variation does not induce any prior impact for this example.
- **c.**✓ Finally, the **causal region** in trace α is highlighted in orange and is mainly due to bus contention.

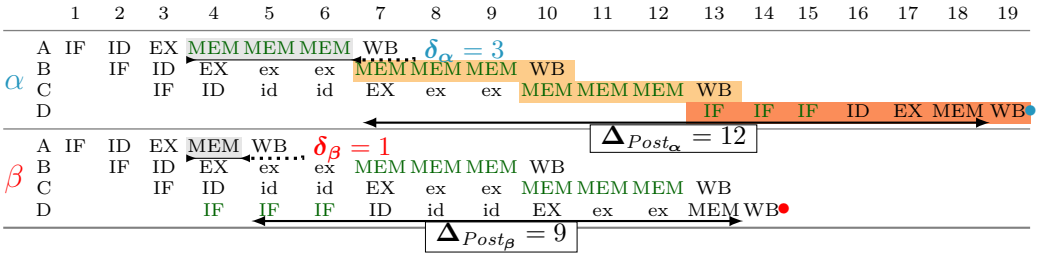


Figure 5 Motivating example with an amplification TA on SIC^-.

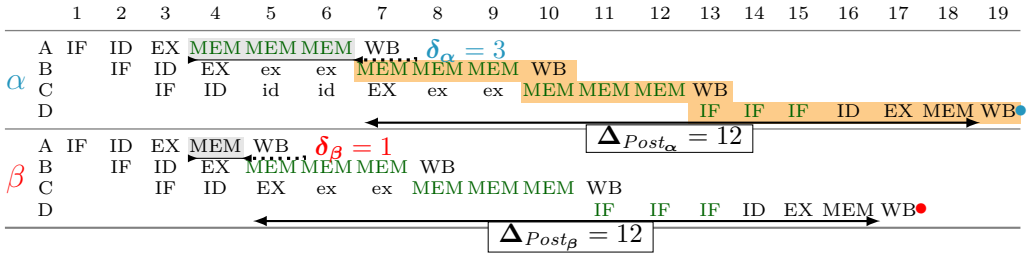


Figure 6 Motivating example avoiding amplification TA on *SIC*.

The execution traces obtained for the same input scenario on *SIC* does not exhibit an amplification due to the counter measures (cf. *mempending*). The resulting traces are shown in Figure 6. Applying the conditions from Definition 2 to this example yields:

- **a.**✓ The same local **variation** occurs as above.
- b.✗ There is no **slowdown** since the relative distance in trace α is the same as in β: $\Delta_\alpha = \Delta_{Post_\alpha} = 12 \not< \Delta_\beta = \Delta_{Post_\beta} = 12$.
- **c.**✓ The **causal region** in trace α remains the same.

Since Condition b is not satisfied, the definition no longer signals an amplification TA in *SIC*. This result is discussed in greater details in Section 6.

5.2 Amplification Example with *Pre-Effect*

We provide a second example that shows an AMP TA for both *SIC* and SIC^-, in which the variation has a prior impact due to the unavailability of the bus.

In Figure 7, we observe that the amplification occurs prior to the variation because the bus is used by instruction A. In SIC^-, when two bus requests occur simultaneously, priority is granted to data accesses in the MEM stage. Consequently, in trace α, the bus access by the IF stage is postponed until the data access is completed, which amplifies the execution time of instruction D.

- **a.**✓ Local **variation** in IF stage of instruction D, satisfying $\delta_\alpha = 3 > \delta_\beta = 1$.
- **b.**✓ **Relative slowdown** in trace α of the entire instruction D, e.g., the release event of the WB stage satisfies $\Delta_\alpha = \Delta_{Pre\alpha} + \Delta_{Post\alpha} = 3 + 3 = 6 > \Delta_\beta = \Delta_{Post\beta} = 5$.
- **c.**✓ Finally, the **causal region** in trace α is due to pipeline stages order (R1.).

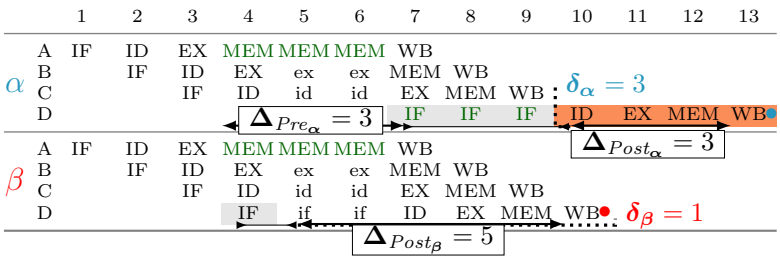


Figure 7 Amplification TA with a *pre-effect* in SIC^-.

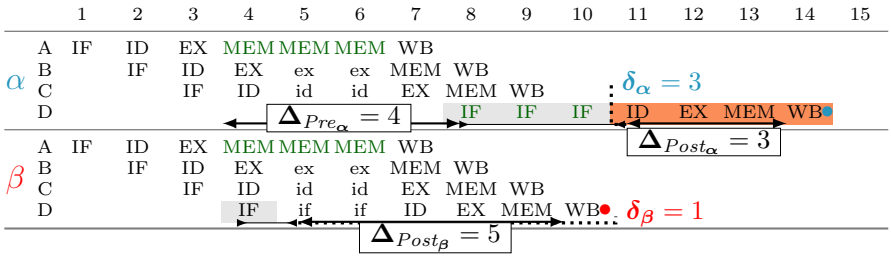


Figure 8 Amplification TA with a *pre-effect* in *SIC*.

A similar behavior is observed in Figure 8. In *SIC*, the fetch of instruction D from the main memory is delayed by one additional cycle compared to SIC^- due to the *mempending* property (since instruction B is a load). Similarly to SIC^-, Definition 2 indicates an AMP TA for this example.

This example demonstrates that *SIC* does indeed exhibit AMP TAs. In the SIC paper [15], the authors address this as a timing compositionality issue and provide a proof for computing a bound for these effects, ensuring that the pipeline remains compositional even with the presence of AMP TAs (and consequently predictable as well). For instance, in this example (assuming a write-through policy for stores), the penalty of the miss on the finishing time in trace α, as provided by the authors, is five times the memory latency (*missLat*) compared to trace β, i.e., 15 cycles.

6 Formal Proofs Regarding *SIC*

Building on the results from Section 6, we prove the absence of CI TAs and of *specific* AMP TA patterns in *SIC*.

To demonstrate this using the CI and AMP definitions previously established, we use a *proof by contradiction*. For this purpose, we consider a pair of execution traces α and β of the same instruction sequence I with the same input data, but starting from distinct initial hardware states. We specifically examine the interactions between two instructions A and $B \in I$, where either of them suffers a variation. Without loss of generality, we assume that A always starts executing before B. Furthermore, we assume that there is *only one variation* between the considered traces, since traces with multiple variations would require reasoning about the *combined effect* of these variations. This is still an open research problem [5]. We refer to this as the ***No Latency Changes (NLC)*** property. A variation may occur at one of the stages IF, MEM, or ST – representing a cache hit in one trace and a miss in the other.

Before presenting the proofs, we first would like to highlight an essential property that emerges from the pipeline behavior. The pipeline exhibits an ***In-order Property (IP)***, meaning that if instruction A starts before instruction B, the relative order of all events $(A, \downarrow s_A, t_A)$ and $(B, \uparrow s_B, t_B)$, where $s_A, s_B \in S \cup S'$ are pipeline stages such that $s_A \sqsubseteq s_B$ ($s_A = s_B$ or s_A is earlier in the pipeline than s_B) [15], must satisfy: $t_A \leq t_B$, where t_A and t_B are respectively the release and the acquisition dates of s_A and s_B.

We use the notation $A.s$ to refer to A's use of stage $s \in S \cup S'$, with $\uparrow A.s = (A, \uparrow s, t_1)$ and $\downarrow A.s = (A, \downarrow s, t_2)$ being the acquisition and release events of s by A (similarly for $B.s$). We also use $A.v$ ($B.v$) to represent the instruction and stage concerned by the variation, where $v \in \{IF, MEM, ST\}$.

6.1 Counter-intuitive TAs in *SIC*

We provide below the proof of absence of CI TAs in *SIC*.

▶ **Lemma 3.** *SIC* ***is free from CI TAs.***

Proof. We proceed by contradiction, assuming that a CI TA occurs in *SIC* and showing that this leads to a contradiction. We first consider the case when a variation of instruction B affects A and then consider the opposite case, i.e., A impacts B.

Variation in B. A CI TA in this case means that:

B1. There is a variation in $B.v$, where B experiences a cache hit in trace α and a cache miss in trace β: $\delta_\alpha < \delta_\beta$.

B2. A slowdown is observed between $B.v$ and $A.s$ in trace α, where s can be any pipeline stage: $\Delta_\alpha = \Delta_{Pre\alpha} + \Delta_{Post\alpha} > \Delta_\beta = \Delta_{Pre\beta} + \Delta_{Post\beta}$.

B3. A causal link exists between $B.v$ and $A.s$ in trace α.

However, Condition 3 leads to a contradiction since:

- A backward dependency between $B.v$ and $A.s$ is impossible, since it does not respect program ordering (recall B is supposed to start before A).
- A resource contention on stages EX/MEM is also impossible due to the IP: the EX/MEM stages must be released by previous instructions before B can acquire them.
- Bus contention is impossible, since $B.v$ in trace α is a cache hit (recall a CI TA implies a favorable case).

A causal link between $B.v$ and $A.s$ can thus be excluded and, consequently, Definition 1 cannot apply.

Variation in A. A CI TA in this case means that:

A1. There is a variation in $A.v$, where A experiences a cache hit in trace α and a cache miss in trace β: $\delta_\alpha < \delta_\beta$.

A2. A slowdown is observed between $A.v$ and $B.s$ in trace α, where s can be any pipeline stage: $\Delta_\alpha = \Delta_{Pre\alpha} + \Delta_{Post\alpha} > \Delta_\beta = \Delta_{Pre\beta} + \Delta_{Post\beta}$.

A3. A causal link exists in trace α between $A.v$ and $B.s$.

From Conditions 2 and 3 it follows that a chain of causal events has to exist where the total length of the chain (Δ_α) is larger than the relative distance in β (Δ_β). Following the IP, NLC, and *mempending* properties, all events between $A.v$ and $B.s$ from trace β should also be in-between $A.v$ and $B.s$ in trace α. Any causal chain due to these events alone cannot be longer in α than in β (NLC property). To satisfy Condition 2, we thus need a new event e that appears on the causal chain and increases its length.

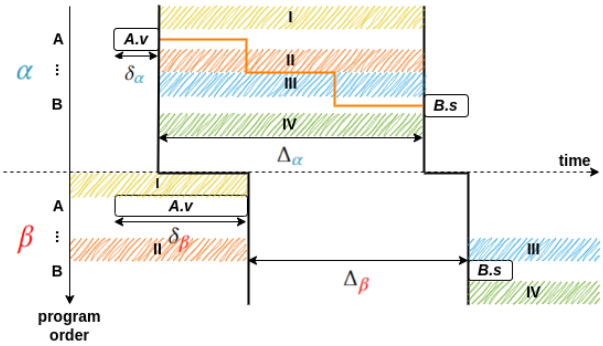


Figure 9 CI TA: Illustration of an event e delaying $B.s$ with regard to a variation $A.v$. Both, e and $B.s$, are on a *causal chain* starting at $A.v$, depicted as an orange line (—). Case I: an event e in the yellow shaded area (▨) is from an instruction starting prior to A, occurring before or in parallel with $A.v$. Case II (▨): e is from an instruction starting after A, executing before or in parallel with $A.v$. Case III (▨): e is from an instruction prior to B, occurring in parallel or after $B.s$. Case IV (▨): e is from an instruction after B, occurring in parallel or after $B.s$.

Figure 9 illustrates how this event e from other instructions (before A, between A and B, or after B) in trace β may become visible in the region between $A.v$ and $B.s$ in trace α and trigger a TA. This figure is provided for $\Delta_{Pre\beta} = 0$, however the proof holds if the variation induces a prior impact in trace β such that $\Delta_{Pre\beta} \neq 0$. We consider each of the four cases (I-IV) illustrated in Figure 9 in order to show that none of these newly appearing events can be on that causal chain.

I If e occurs before $A.v$, its position does not change due to the IP and NLC properties. If e is in parallel with $A.v$ in β, it can happen after $A.v$ in α (since $A.v$ finishes earlier) but e cannot be causal to $A.v$ (same proof as a variation in B). Therefore, since e is not in the causal chain, it does not impact Δ_α.

II For $v = IF$: There cannot exist an event e, occurring before or in parallel with $A.IF$ in trace β. This is ensured by the IP, since this e would have to be preceded by a new fetch, which may only start after the end of $A.IF$.
For $v = MEM$: If e happens before $A.MEM$, it does not move (IP and NLC). However, if e occurs in parallel with $A.MEM$, it can only concern stages $\{IF, ID, EX\}$ (involving up to 3 instructions), and can move between $A.MEM$ and $B.s$ in α. In this case, a causal link is not possible between $A.MEM$ and e in trace α since: bus contention between $A.MEM$ and IF cannot happen because $A.MEM$ is a hit. A data dependency between $A.MEM$ and EX is also excluded, since it would still hold in trace β and thus cannot be a new event. Thus e is not on the causal chain and does not impact Δ_α.
For $v = ST$: The same argument as for $v = MEM$ applies.

III In trace β, e happens in parallel or after $B.s$, which means that $B.s$ has no dependency with e. It also means that e concerns a stage s' such that $s \sqsubset s'$ (IP). Thus, $B.s$ cannot experience resource contention with e in trace α as well. Finally, bus contention is also not possible between e and $B.s$, since otherwise the access order to the bus would not have been maintained between the two traces due to the *bus arbiter*. Therefore, e is not on the causal chain in α and its timing does not impact Δ_α.

IV In trace β, e concerns a stage s' such that $s \sqsubseteq s'$. Following the IP, e cannot advance before $B.s$ in trace α.

Consequently, **no CI TA is possible in** *SIC*. ■ ■ ■ ■ ■ ■ ■ ◀

6.2 Amplification TAs in *SIC*

As shown in the experiments, *SIC*, in fact, does exhibit AMP TAs. However, we observe that none of the AMP TAs detected in our experiments exhibits a *pre-effect*, i.e., $\Delta_{Pre\alpha} = 0$. So, while it is not possible to prove the absence of all AMP TAs, it is indeed possible to also formally prove the absence of this class of AMP TAs.

▶ **Lemma 4.** *SIC* ***is free from AMP TAs if*** $\boldsymbol{\Delta_{Pre\alpha} = 0}$.

Proof. We proceed in a similar way as for Lemma 3. We assume that an AMP TA occurs and, considering the same case analysis from before, show that this leads to a contradiction.

Variation in B. An AMP TA in this case means that:

B1. There is a variation in $B.v$, where B experiences a cache miss in trace α and a cache hit in trace β: $\delta_\alpha > \delta_\beta$.

B2. An over-proportional slowdown is observed between $B.v$ and $A.s$ in trace α, where s can be any pipeline stage: $\Delta_\alpha = \Delta_{Post\alpha} > \Delta_\beta = \Delta_{Post\beta}$.

B3. A causal link exists between $B.v$ and $A.s$ in trace α.

In Lemma 3, we demonstrated in this similar case that a causal link between $B.v$ and $A.s$ is not possible, whether through a dependency or contention on the EX/MEM stages. The same proof holds.

However, unlike in Lemma 3, δ_α represents a cache miss, meaning we must consider potential bus contention. Given the behavior of the *bus arbiter*, instructions starting prior to B always acquire the bus before B when needed. Consequently, no causal link exists from $B.v$ to $A.s$.

Note: For *SIC* $^-$, such a causal link is indeed possible, since $B.IF$ could acquire the bus before $A.MEM$ in some situations. this behavior is one reason for having such AMP TAs in *SIC* $^-$.

As before, it follows that Condition 3 cannot be satisfied, which excludes AMP TAs in this case.

Variation in A. An AMP TA in this case means that:

A1. There is a variation in $A.v$, where A experiences a cache miss in trace α and a cache hit in trace β: $\delta_\alpha > \delta_\beta$.

A2. An over-proportional slowdown is observed between $A.v$ and $B.s$ in trace α, where s can be any pipeline stage: $\Delta_\alpha = \Delta_{Post\alpha} > \Delta_\beta = \Delta_{Post\beta}$.

A3. A causal link exists between $A.v$ and $B.s$ in trace α.

We proceed similarly to Lemma 3, that is from Conditions 2 and 3 it follows that a chain of causal events has to exist where the total length of the chain (Δ_α) is larger than the relative distance in β (Δ_β).

Following the IP, NLC, and *mempending* properties, all events between $A.v$ and $B.s$ from trace α should also be in-between $A.v$ and $B.s$ in trace β. Any causal chain due to these events alone cannot be longer in α than in β due to the IP and NLC properties. To satisfy Condition 2, we thus need a new event e that appears on the causal chain and delays $B.s$.

We illustrate in Figure 10 how this event e from other instructions (before A, between A and B, and after B) in trace β may move between $A.v$ and $B.s$ in α and amplify this delay. We consider once again each of the four cases (I-IV) illustrated in Figure 10 to show that none of these newly appearing events can be on that causal chain.

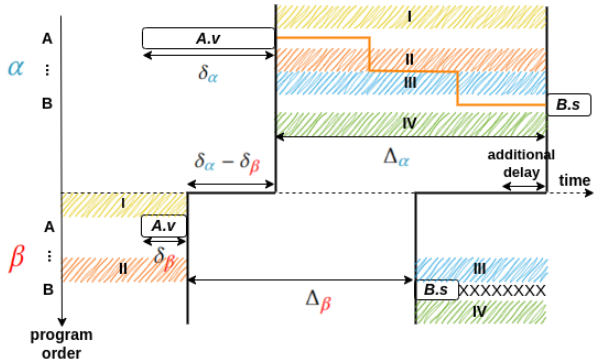


Figure 10 AMP TA: Illustration of an event e delaying $B.s$ with regard to a variation $A.v$. Both, $B.s$ and e are on a *causal chain* starting at $A.v$, depicted as an orange line (—). Case I (▨): e is from an instruction prior to A, occurring before or in parallel with $A.v$. Case II (▨): e is from an instruction after A, occurring before or in parallel with $A.v$. Case III (▨): e is from an instruction prior to B, occurring in parallel or after $B.s$. Case IV (▨): e is from an instruction after B, occurring in parallel or after $B.s$.

I We know from Condition 1 that $\delta_\beta < \delta_\alpha$. Following the IP and NLC properties, e can never occur after $A.v$ in trace α.

II For $v = IF$: As before, e has to be preceded by an instruction fetch, which cannot start before the end of $A.IF$.
For $v = MEM$: Following the IP and NLC properties, since e happens before or in parallel with $A.MEM$ it cannot move. We emphasize that if e indicates a fetch miss, then it will not be allowed to enter the pipeline until the end of $A.MEM$ due to the *mempending* property (even if $A.MEM$ is a hit). This means that e in this case will come after $A.MEM$ in both traces.
For $v = ST$: The same argument as for $v = MEM$ applies.

III The same arguments as in Lemma 3, Case III apply.

IV The same argument as in Lemma 3, Case IV applies.

Consequently, **no AMP TA is possible in** *SIC* **if** $\boldsymbol{\Delta_{Pre\alpha} = 0}$. ■■■■■■■ ◀

7 Discussion and Future Work

In this paper, we proposed a study of CI and AMP TAs in in-order pipelines. In Sections 4- 6, we have provided a definition of CI TAs by building upon and revising an existing causality-based definition [5], ensuring that it is applicable to the studied in-order pipelines. We subsequently proposed a new definition for AMP TAs and we presented our experiments and evaluation of those definitions through automated tools. We thus addressed the three questions motivating our study.

While the proposed research provides a way to reason about the absence/presence of timing anomalies in these pipelines, it does have some limitations. One such limitation is that the causality concept is architecture-specific. Although the definitions are general and applicable to multiple architectures, the construction of the ETDG – and, consequently, reasoning about causality – depends on architectural details. This can be seen as an advantage, as it allows for more precise detection of TAs that are closely tied to the specific characteristics and behavior of the architecture. However, it also implies that adaptations are necessary when applying the definitions to different architectures.

Additionally, our approach does not fully account for all architectural features, such as branch prediction or pipeline buffers, et cetera, which may contribute to the occurrence of TAs. These elements are left for future work.

Despite these limitations, the definitions we provide offer a key advantage in addressing both predictability and performance, as well as the trade-off between them. First, these definitions are highly beneficial in the design of provably predictable architectures, as they allow for more confidence in the absence of TAs, rather than relying on conditions that are merely considered sufficient for their absence. This advantage is demonstrated by the definitions and proofs we provided for the SIC core. Additionally, the detection procedures help identify TA patterns on architectures, enabling a more flexible design approach (i.e., an approach that allows predictability under specific constraints, while also accounting for performance), rather than strict conditions like imposing a monotonic cycle behavior. Finally, our work provides a foundation for addressing TAs, offering insights that can guide the development of mechanisms to handle them, potentially enhancing performance while maintaining predictability. For example, consider the case presented in Figure 8. As previously stated, the WCET bound from the SIC paper predicts an additional 15 cycles for the behavior in trace α compared to trace β. However, our analysis shows that the actual bound is only 4 cycles, indicating that the estimate from the SIC paper is overly pessimistic in some cases. Our approach, associated with mechanisms that handle TAs based on the obtained results, would enable more flexible timing analysis which would lead to a better balance between predictability and performance. A more detailed study of TA patterns, as well as the identification and implementation of the stated mechanisms represent key areas for future work.

References

1 Sahar Abbaspour, Florian Brandner, and Martin Schoeberl. A Time-Predictable Stack Cache. In *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, pages 1–8, 2013. doi:10.1109/ISORC.2013.6913225.

2 Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of Resource Sharing on Performance and Performance Prediction: a Survey. In *Proceedings of the 24th International Conference on Concurrency Theory*, pages 25–43. Springer-Verlag, 2013. doi:10.1007/978-3-642-40184-8_3.

3 Infineon Technologies AG. TriCore 1 Pipeline Behaviour and Instruction Execution Timing, 2004.

4 Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. In *2011 IEEE 32nd Real-Time Systems Symposium (RTSS)*, pages 261–271, 2011. doi:10.1109/RTSS.2011.31.

5 Benjamin Binder, Mihail Asavoae, Florian Brandner, Belgacem Ben Hedia, and Mathieu Jan. The Role of Causality in a Formal Definition of Timing Anomalies. In *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 91–102, 2022. doi:10.1109/RTCSA55878.2022.00016.

6 Benjamin Binder, Mihail Asavoae, Belgacem Ben Hedia, Florian Brandner, and Mathieu Jan. Is This Still Normal? Putting Definitions of Timing Anomalies to the Test. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 139–148, 2021. doi:10.1109/RTCSA52859.2021.00024.

7 Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. What is a Timing Anomaly? In *12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012)*, volume 23, pages 1–12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/OASIcs.WCET.2012.1.

8 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Handbook of Model Checking. *Springer*, 1:1–1200, 2018.

9 Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A Method Cache for Patmos. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 100–108, 2014. doi:10.1109/ISORC.2014.47.

10 Stephen A. Edwards and Edward A. Lee. The Case for the Precision Timed (PRET) Machine. In *Proceedings of the 44th Annual Design Automation Conference (DAC)*, pages 264–265. Association for Computing Machinery, 2007. doi:10.1145/1278480.1278545.

11 Gernot Gebhard. Timing Anomalies Reloaded. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 1–10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010. doi:10.4230/OASIcs.WCET.2010.1.

12 R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969. doi:10.1137/0117039.

13 Alban Gruin, Armelle Bonenfant, Thomas Carle, and Christine Rochange. Modelling and Proving the Monotonicity of Processor Pipelines in Coq. In *2024 22nd ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE)*, pages 12–21, 2024. doi:10.1109/MEMOCODE63347.2024.00007.

14 Alban Gruin, Thomas Carle, Hugues Cassé, and Christine Rochange. Speculative Execution and Timing Predictability in an Open Source RISC-V Core. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 393–404, 2021. doi:10.1109/RTSS52674.2021.00043.

15 Sebastian Hahn and Jan Reineke. Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 469–481, 2018. doi:10.1109/RTSS.2018.00060.

16 Sebastian Hahn and Jan Reineke. Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core. *Real-Time Syst.*, 56(2):207–245, 2020. doi:10.1007/s11241-019-09341-z.

17 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Toward Compact Abstractions for Processor Pipelines. In *Correct System Design, September 8-9, 2015, Proceedings*, pages 205–220. Springer International Publishing, 2015. doi:10.1007/978-3-319-23506-6_14.

18 Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards Compositionality in Execution Time Analysis: Definition and Challenges. *SIGBED Rev.*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

19 Ranjit Jhala and Kenneth L. McMillan. Microarchitecture Verification by Compositional Model Checking. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, pages 396–410. Springer-Verlag, 2001. doi:10.1007/3-540-44585-4_40.

20 D. Kröning. *Formal Verification of Pipelined Microprocessors*. Phd thesis, Universitat des Saarlandes, Saarbrücken, Germany, 2001.

21 Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

22 lowRISC. Ibex: An Embedded 32-bit RISC-V CPU Core. https://www.lowrisc.org/ibex, 2020.

23 T. Lundqvist and P. Stenstrom. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 12–21, 1999. doi:10.1109/REAL.1999.818824.

24 Kenneth L. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV)*, pages 110–121. Springer-Verlag, 1998. doi:10.1007/BFB0028738.

25 Michael Platzer and Peter Puschner. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196, pages 1:1–1:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECRTS.2021.1.

26 Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation. In *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108, 2011. doi:10.1145/2039370.2039388.

27 Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4, pages 1–6. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2006. doi:10.4230/OASIcs.WCET.2006.671.

28 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, and Christian W. Probst. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES)*, volume 18, pages 11–21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. doi:10.4230/OASIcs.PPES.2011.11.

29 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008. doi:10.1145/1330611.1330612.