

Formal Modeling of Multi-Core Systems to Investigate Causality in Timing Anomalies

Abstract. Multi-core systems are widely used in modern real-time systems because they offer higher performance than single-core architectures. The presence of Timing Anomalies (TAs), can affect timing analysis and consequently compromise the predictability of real-time systems. TAs have been widely studied for processor cores or cache replacement policies, but research on TAs in multi-core systems is considerably less developed.

In this work, we use Time Petri Nets (TPNs) to investigate counter-intuitive TAs in bus arbitration. More precisely we investigate the notion of *causality*, inherent to TPNs, and its connection with a recently proposed definition based on temporal causality of counter-intuitive TAs. Starting from a regular TPN model of the arbiter and the computation of cores, we show that an equivalent model that explicitly tracks causality can be derived using 3 simple transformation steps. Temporal causality propagates naturally through this model, which can be used to formally characterize TAs in a multi-core context.

Keywords: Real-Time Systems · Multi-Cores · Timing Anomalies · Causality · Time Petri Nets.

1 Introduction

The high performance of multi-core systems stems from their ability to execute multiple programs in parallel. However, when several cores share common resources – such as main memory, caches, and buses with concurrent accesses being managed by an arbitration mechanism (Round-Robin, Time-Division Multiplexing, First-Come First-Served) – this sharing may introduce execution time dependencies between cores: a core accessing a shared resource may impact the execution time of other cores. The resulting interference generates delays that propagate through the system, altering the timing of subsequent resource accesses, thereby affecting future interference conditions and potentially even leading to *Timing Anomalies* (TAs). TAs may render conventional timing analysis approaches unsafe, *e.g.*, when an analysis assumes *compositionality* [11] and tries to add up timing effects stemming from seemingly independent resources. TAs may invalidate this assumption and consequently lead to underestimation. This makes it difficult to ensure timing predictability required by real-time systems.

TAs were first discovered by Lundqvist and Stenström [17] on dynamically scheduled out-of-order processors. They identified two broad forms: (i) counter-intuitive effects, where a locally faster event leads to a globally slower execution, and (ii) amplification effects, where a locally slower event leads to

an even larger slowdown globally. Subsequent formal definitions and classifications of TAs [20,9,6,2], mostly focused on the processor level, including caches and pipelines. To the best of our knowledge, only Shah et al. [22] investigated TAs beyond the processor-level in Round-Robin (RR) arbitration. They discovered counter-intuitive TAs using two contention metrics: aggressiveness and α -interference. Unfortunately, these metrics are defined only informally and thus remain open to interpretation.

More recently, Binder et al. [3] proposed a new approach to defining counter-intuitive TAs, identifying causality as the key ingredient missing in prior work to establish relations between local and global timing effects. From resource acquisition and release events occurring in an out-of-order processor they construct an *Event Time-Dependence Graph* (ETDG), capturing the impact of an event on the timing of future events. The nodes and arcs of the graph represent timestamped events and the minimal delay imposed between two events respectively. While the graph allows to formalize the timing relations between events and *temporal causality*, its construction is ad hoc. Arcs are derived from a set of micro-architecture-dependent rules that capture expert knowledge on the interactions between the processor’s resources, but have no formal foundation. Consequently, there is no formal guarantee that all relevant timing dependencies are correctly captured, i.e., rules might be wrong, incomplete, or missing.

In this work, we aim to formalize TAs in multi-core systems based on the definition of Binder et al. [3]. Our formal model uses High-Level Colored *Time Petri Nets* (TPNs) [12] to implement a Round-Robin (RR) bus arbiter and alternating sequences of computation and memory access phases for each core. Tokens of a given color represent memory requests emitted by a core associated with that color. The tokens circulate through the different hardware components: from the core to the arbiter, through the memory, and back to the core. Those components are represented as places in the model and each place may hold possibly multiple tokens of different colors to model intrinsic processing or extrinsic blocking *delays*. The movement of tokens between places at a given *time instant* is governed by transitions, which enforce correct system behavior.

In order to reason about TAs, *temporal causality*, as defined by Binder et al. [3], has to be encoded in our formal model. For this, we exploit the inherent notion of causality in TPNs, which allows us to do away with the ad-hoc construction rules. The movement of tokens through the underlying TPN gives rise to a graph that is structurally almost equivalent to the ETDG. Based on this observation, we encode temporal causality by transforming the original TPN model. The transformation proceeds according to three possible actions: (i) creating temporal causality, (ii) preserving temporal causality, and (iii) breaking temporal causality when the time a token spends in a place is *delayed due to extrinsic blocking*. Temporal causality then emerges directly from the flow of tokens in the transformed TPN model, allowing us to reason about TAs in a multi-core setting through model checking and parameter synthesis as provided by the Roméo [8,16] framework.

The rest of the paper is organized as follows. Section 2 provides additional details on the definition of Binder et al. [3] and TPNs. Section 3 presents the system model and its implementation as a TPN. Section 4 provides an in-depth discussion of the causality notion inherent to TPNs and its relationship to the causality notion in TAs. Section 5 presents the causality-enhancement transformation and some correctness arguments. Section 6 applies this transformation to identify a counter-intuitive TA in Round-Robin arbitration. Section 7 discusses prior work leading up to the conclusion in Section 8.

2 Background

This section covers two ingredients of our approach: the causality-based definition of counter-intuitive TAs in Section 2.1 and the formal foundations of Time Petri Nets in Section 2.2.

2.1 Defining Timing Anomalies Through Temporal Causality

The most recent definition of TAs at the processor-level was proposed by Binder et al. [3]. Compared to previous attempts to define TAs [24,9,14,6], the proposed approach appears more consistent in classifying similar TA patterns in different contexts due to the novel notion of *temporal causality*. Consequently, it appears to be a good starting point for our work to characterize TAs in a multi-core setting.

The definition formally characterizes counter-intuitive TAs and a corresponding decision procedure has been implemented in a tool. The formalization is inspired by the intuitive notion of TAs, where a *local speedup* leads to a *global slowdown*, and compares two executions of the same program (α and β) starting from possibly different initial hardware states. A TA is detected if the following three conditions are satisfied:

- **Favorable variation:** A variation indicates a difference in the latency $\delta_\alpha/\delta_\beta$ associated with the usage of a resource by an instruction in the two executions, respectively. The execution with the smaller latency is called *favorable*, e.g., $\delta_\alpha < \delta_\beta$ indicates a favorable variation in α . This characterizes the intuitive notion of a local speedup, which marks the potential starting point of a counter-intuitive TA.
- **Slowdown:** Contrary to most previous definitions the global slowdown is expressed through a *relative* time delay $\Delta_\alpha/\Delta_\beta$ from the end of the respective variation in the two executions with regard to a pair of *corresponding* events, e.g., the completion of the same instruction in both executions. When the favorable execution experiences a longer delay, e.g., $\Delta_\alpha > \Delta_\beta$, a slowdown is observed that could potentially indicate a TA.
- **Causality:** The novel notion that distinguishes Binder et al.’s [3] definition is temporal causality, which establishes a chain of events starting at the favorable variation and delaying each other up to a corresponding event that

experiences a relative slowdown. This notion of causality establishes that the slowdown is in fact caused by the favorable event and not by another, unrelated, event.

The notion of *temporal causality* in the definition aims to identify chains of events that successively delay each other. An execution is viewed as a set of timestamped events, such as the acquisition or the release of a resource (a pipeline stage, a functional unit) by an instruction. Events are not independent: an event can only occur after some other events, and at least a minimal delay later (e.g., an instruction leaves a stage only after spending its latency in it, and acquires a resource only once it has been released). An event typically depends on several earlier events, and its timestamp is set by the one that constrains it the most. Formally, if an event e_2 depends on an event e_1 with a minimal delay w , then $t_2 \geq t_1 + w$; the dependency is said to be *causal* when $t_2 = t_1 + w$, i.e., when e_1 *determines* the timestamp of e_2 .

Causality is transitive: a chain of causal dependencies propagates the timing of its first event to all subsequent ones. Starting from an event subject to a local timing variation, the events reachable through such chains form the *causal region* of the variation: these are exactly the events whose timestamps are governed by the variation. Conversely, causality is lost as soon as an event waits for another event which is not part of the chain of causal dependencies and that constrains it more.

To apply this definition, the timing dependencies between events, and their minimal delays, must be known. In [3], they are derived by hand from the authors' knowledge of the out-of-order pipeline under study. Such a derivation offers no formal guarantee that all relevant dependencies are captured, and it is tightly coupled to that specific architecture: it is unclear how it could be adapted to other contexts, such as multi-core systems in which dependencies also arise between cores through shared buses and their arbitration.

2.2 Time Petri Nets

We use *Time Petri Nets* (TPNs) extended with colors, parameters, and finite variables, as implemented in the Roméo framework [16]. In the following, we recall a formal definition of TPNs, adapted from previous work [16], before describing the modeling and verification capabilities of Roméo that we rely on in this paper.

Definition 1. A colored parametric TPN is a tuple $N = (P, T, C, Var, pre, post, Is, g, u, m_0)$ where:

- P and T are finite, disjoint sets of places and transitions,
- C is a finite set of colors, used to distinguish tokens,
- Var is a finite set of variables, ranging over integers and colors,
- pre and $post$ are backward and forward incidence functions, mapping each transition to the (colored) tokens it consumes from its input and produces in its output places respectively,

- I_s is the static firing interval function, associating each transition with a fixed lower and upper time bound (its earliest and, possibly infinite, latest relative firing time),
- g is the guard function, associating each transition with a Boolean-valued function over Var and the current marking (*i.e.*, mapping of colored tokens to places),
- u is the update function, associating each transition with a function that updates the values of Var and the marking when the transition fires,
- m_0 is the initial marking.

Intuitively, a colored TPN model consists of variables, places, and transitions. Each place can hold colored tokens, which may move between places through the firing of transitions. A marking indicates, for each place, the tokens (and their colors) it currently holds and for each variable its value. A transition is enabled once its input places hold enough (colored) tokens, as given by pre , and its guard g evaluates to true (note that g may inspect both variables and the number/colors of the tokens present in its input places). Each transition is associated with a *clock* that is running whenever the transition is enabled and reset whenever the transition is not enabled. An enabled transition may only fire if the value of its clock is within the static firing interval, triggering a number of actions: removing the tokens specified by pre from its input places, producing the tokens specified by $post$ in its output places, updating the values of the variables with the update function u , and updating clocks.

In the context of our work, the parametric extension of TPNs allows the static firing interval $I_s(t)$ of a transition t to refer not only to fixed bounds but also to parameters, whose values are constant but left unspecified within a potentially infinite domain. Several parameters can also be linked together through linear constraints. Specifying an interval $[a, a]$ for a transition, where a is a parameter defined over the domain $[a_{min}, a_{max}]$ (where a_{min} and a_{max} are rational numbers), allows us to model an event with a constant but unknown latency. This is not equivalent to using a transition with an interval $[a_{min}, a_{max}]$, which allows you to model an event with variable latency. It is of course possible to mix parameters and numbers, to model for instance an event with an unknown constant latency but subject to a variable jitter, e.g., $[a - j_{min}, a + j_{max}]$ with j_{min} and j_{max} rational numbers. The colored extension allows tokens of different colors to flow through the same places and transitions while remaining distinguishable: a guard or update function may refer to the color of the token going through the transition using the variable **\$any**. The colored extension is syntactic sugar, as a semantically equivalent model can be obtained by unfolding techniques [13].

In this work we use Roméo, a formal modeling and verification tool for systems specified as TPNs in the sense of Definition 1. Roméo provides capabilities for exploring the state space of a model either through simulation or model checking. The model checker takes as input a TPN model, an initial state, and a temporal logic formula and exhaustively explores a symbolic abstraction of the infinite state space to determine whether the model satisfies the given formula. Roméo can provide witness traces as sequences of timestamped transition

firings. When the model contains parameters, Roméo also provides the domain of values, defined as a set of linear constraints, for which the formula holds. In general, this problem is undecidable: there is no algorithm that is guaranteed to give an answer in finite time for every parametric TPN, because the parameters can take infinitely many values and the analysis may never end. The problem becomes decidable for a restricted class, called L/U, in which each parameter is used either only as a lower bound or only as an upper bound of the firing intervals [23]. Outside this class, Roméo relies on semi-algorithms: they may not terminate, but when they do, the computed domain of parameter values is exact.

3 Modeling Round-Robin Arbitration in Multi-Cores

We consider a system of N identical cores $core_c$, $c \in \{0, \dots, N-1\}$, connected to a shared main memory via a common bus. The execution of $core_c$ is modeled as a finite sequence of alternating computation and memory access phases $\langle C_{0,c}, C_{1,c}, \dots \rangle$, where each $C_{i,c} \in \mathbb{N}^{\geq 0}$ represents a computation duration. A memory access request $R_{i,c}$ is emitted at the end of each computation phase $C_{i,c}$. A core that has issued a request is blocked until the request completes, meaning no core can have more than one request in flight at any given time. We consider only fixed-size memory accesses.

Bus access is managed by a Round-Robin (RR) arbiter operating over a logical ring of cores. Upon contention, the arbiter grants the bus to the first pending core encountered when scanning the ring from the current priority holder. Priority is then transferred to the immediate successor of the served core, guaranteeing starvation-free arbitration. A granted request blocks the bus for a constant service time T_{mem} , after which the bus becomes available for the next request.

Since requests are emitted at the end of computation phases, we denote by $EC_{i,c}$ the time at which request $R_{i,c}$ is emitted. The first request of each core arrives at:

$$EC_{0,c} = C_{0,c} \quad (1)$$

For $i > 0$, the arrival time satisfies the recurrence:

$$EC_{i,c} = EC_{i-1,c} + L(R_{i-1,c}) + C_{i,c}, \quad (2)$$

where $L(R_{i,c})$ is the latency experienced by request $R_{i,c}$, defined as:

$$L(R_{i,c}) = B(R_{i,c}) + T_{\text{mem}}. \quad (3)$$

The term $B(R_{i,c})$ captures the time a request is blocked before being granted bus access and thus depends on contention from other cores.

We will now present a high-level colored TPN that implements this system model, and which can be used to compute $L(R_{i,c})$. Figure 1 provides an overview. It consists of two interacting components: (i) a RR bus arbiter (yellow square) and (ii) a request generator (white square). Following standard TPN notation, places are depicted as circles (○) and transitions as rectangles (■). Guards and updates

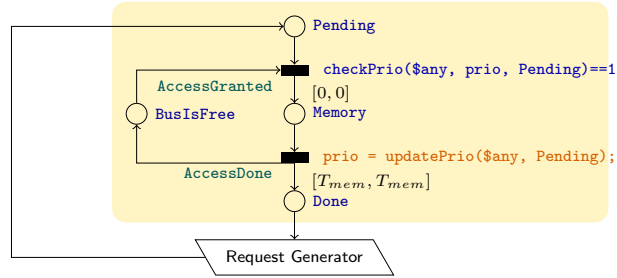


Fig. 1: RR arbiter (yellow) with a placeholder for a request generator (parallelogram).

of transitions are annotated in blue and red, respectively. Tokens represent pending memory requests, where the token color identifies the originating core. The arbiter component is comprised of three places, namely **Pending**, **Memory**, and **Done**, along with two transitions, **AccessGranted** and **AccessDone**.

Tokens residing in **Pending** represent requests that have been emitted by the core, but not yet granted bus access. The transition **AccessGranted** fires when two conditions are met: the bus is idle (token in **BusIsFree**) and the function **checkPrio** confirms that the core is the next in line for service. Specifically, **checkPrio** takes as input the color of a candidate request r ($\$any$), the color of the highest-priority core ($prio$), and the set of tokens currently in **Pending**. It returns true (1) if and only if r holds the current priority. Upon firing, tokens move from **Pending** and **BusIsFree** to **Memory**, indicating that the request has been admitted to the bus.

A token in **Memory** remains there for exactly T_{mem} time units, reflecting the fixed duration of a memory transaction. At the end of the memory access, the transition **AccessDone** fires, transferring the token to **Done** and **BusIsFree**, releasing the bus, and advancing the priority to the successor core (**updatePrio**). A token reaching **Done** may then be consumed by the request generator, which uses it to trigger the next computation phase on the corresponding core.

The **Request Generator** serves a dual purpose: it models the computation phases of each core and drives the emission of memory requests. Whenever a core $core_c$ finishes a computation phase of duration $C_{i,c}$, it emits a request by moving a token to **Pending**. It then waits for a token of the core's color to appear in **Done**, which signals that the core's last memory access has completed and that the core may resume with its next computation phase. The tokens/requests consequently circulate between the core, arbiter, memory, and back to the core. Note that the request generator represents any possible sequence, through a loop within the TPN, or may represent a specific (potentially parameterized) execution scenario, through an explicit sequence.

Figure 2 illustrates a request generator for a specific execution scenario for a system with two cores. Each core alternates between execution and memory requests. In this scenario, it is assumed that the latency of execution phases are constants: $\langle 5, 20 \rangle$ for $core_0$ and $\langle 0, 0, 10 \rangle$ for $core_1$. The resulting execution is

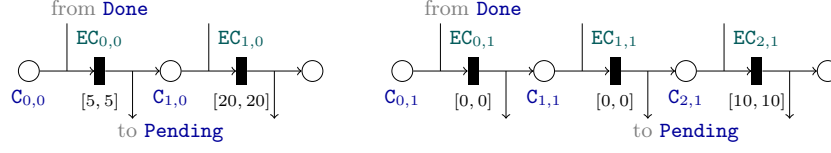


Fig. 2: A request generator for two cores, emitting two/three requests each.

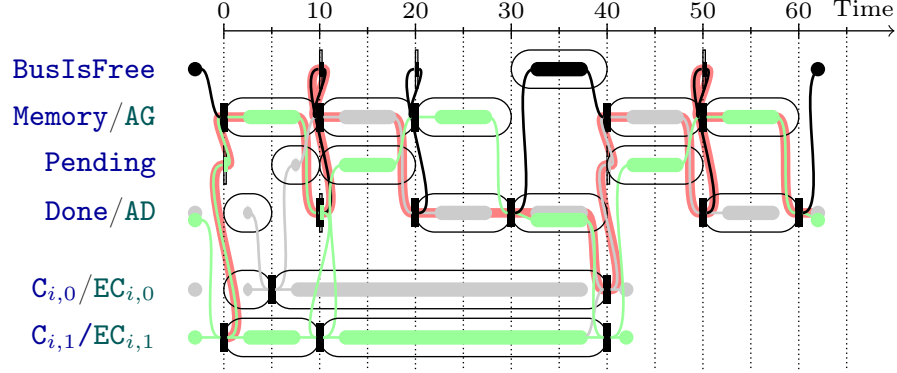


Fig. 3: Execution according to the request generator from Figure 2 .

depicted in Figure 3. The rows represent places holding at least one token of the indicated colors in the initial/final state ($\bullet$) and over time ($\blacksquare$) along with transitions ($\blacksquare$). Arrows indicate the movement of tokens between places due to the firing of transitions. For instance, at time instant 5 transition $EC_{0,0}$ fires, moving tokens from **Done** and $C_{0,0}$ to **Pending** and $C_{1,0}$.

4 Analyzing Causality in TAs with TPNs

Causality emerges naturally from the token flow in TPNs. A transition cannot fire without a token in its input place, and that token can only be there because some earlier transition placed it, which directly encodes a causal dependency between the two transitions. Figure 3 illustrates such dependencies as a red chain: each red segment connects the firing of one transition to the firing of the next transition it enables, by following the token as it moves from an output place of the former to an input place of the latter. For instance, the red chain crosses from **Pending** to **Memory** at each firing of **AccessGranted** (AG): the memory access could not have occurred without the token previously deposited in **Pending**. From any state, it is thus possible to follow back the flow of tokens and recover the full history of the transitions that produced them. We refer to this notion as *TPN causality*, as it emerges directly from the firing semantics of the TPN, without requiring any additional rule.

TPN causality, however, does not coincide with *temporal causality* as defined by Binder et al. [3]. Informally, an event is temporally causal for a later event if it fully determines its date: the delay between the two events is entirely due to the activity that separates them, with no additional waiting introduced by another event. In terms of tokens, this depends on *why* a token stays in a place. A token in **Memory** waits for the fixed access time T_{mem} , and a token in a computation place $C_{i,c}$ waits for its computation time: in both cases, the delay is an intrinsic property of the activity, and temporal causality is preserved. A token in **Pending**, on the contrary, may be blocked by a request of another core currently accessing the memory.

This difference is visible in Figure 3. The request issued by $EC_{2,1}$ at $t = 40$ stays in **Pending** until $t = 50$. TPN causality links this request to its access, since the access could not start without the token in **Pending**. Yet the date of the access ($t = 50$) is not determined by the core that issued the request, but by the end of the access of the other core, which releases the bus at $t = 50$. From the point of view of temporal causality, the chain coming from the issuing core is therefore broken at this point, while the chain coming through the bus (**BusIsFree**) continues. TPN causality thus captures more dependencies than temporal causality: to recover the latter, causal chains must be cut wherever a token is blocked rather than making progress. Section 5 shows how the model is transformed to do so, using additional colours and dropping transitions.

5 Propagation of Temporal Causality

As highlighted in the previous section, the notion of temporal causality is almost perfectly captured by the causality notion that emerges from the movement of tokens in a TPN. However, there are two issues. First, causality in TPN covers all movements of tokens, whereas temporal causality is only concerned with specific events that correspond to favorable variations, as introduced in Section 2. The second problem is that causality in TPNs does not distinguish between time spent in a place due to *extrinsic blocking* (e.g., waiting for the bus to be free), and time spent due to the *intrinsic duration* of the activity associated with the place. This distinction matters: blocking can break temporal causality at places where TPN causality alone would not do so. For these reasons, we need to track temporal causality explicitly.

To address these two problems, we explicitly track temporal causality through token colors. We transform the initial TPN model of Figure 1 so that temporal causality is propagated as token movement: colors are used to distinguish between causal and non-causal tokens, allowing the model itself to encode which token movements belong to the causal chain of the favorable variation and which do not. As mentioned in the system model in Section 3, tokens represent memory requests and colors in the range $\{0, \dots, N - 1\}$ identify the originating core. To track how temporal causality propagates, we double the colors: for each core, one color represents its non-causal tokens ($\{0, \dots, N - 1\}$) that lie outside the causal region of a favorable variation, and a second color represents its causal

tokens $(\{N, \dots, 2 \cdot N - 1\})$, encoding the events that lie within the causal region of the favorable variation. We encode causality as a color on tokens rather than with variables, because causality is a property of each request: several requests may be in progress at the same time, each with its own causal status. With colors, this status is carried by the token and follows it through the net automatically.

Three steps are required to build the transformed model: first, identifying where temporal causality begins and which transition triggers its creation; second, describing how causal and non-causal tokens propagate through the model as transitions fire; and third, determining where and how temporal causality can break that is, which transitions cause a causal token to lose its causal status. The remainder of this subsection addresses these three aspects in turn.

Creating Causal Tokens: As mentioned in Section 2, the causality chain begins at a favorable variation. In our model, this check is performed directly within the request generator, by comparing the computation time of a request in one execution to the computation time of the corresponding request in a reference execution: a shorter computation time identifies a favorable variation. The favorable variation occurs at the transition where this shorter computation time is observed. The causal chain of interest originates at this variation: it identifies the local speed-up whose propagation through the model we want to trace, in order to determine whether it induces a counter-intuitive slowdown elsewhere. In the initial state, all tokens in the model are non-causal. Once such a transition fires, a causal token is produced and flows into place **Pending**, marking the start of the causal chain.

Propagating (Non-)Causal Tokens: The remaining transitions of our TPN model have to be transformed in order to correctly handle the movement of causal and non-causal tokens. The transformation is rather mechanical and is probably amenable to automation (though we do not pursue this here): each transition is duplicated a number of times in order to cover all possible combinations of causal/non-causal tokens at the source places of the transition. When all the tokens at the source places of the transition are non-causal the output tokens have to be non-causal as well. In all other cases at least one of the tokens is causal and consequently the output tokens become causal. Note that Roméo cannot compute the color of tokens produced by a transition dynamically, the only solution is thus to enumerate all possible cases. For instance, the **AccessDone/AD** transition in the original model moves a token from **Memory** to **Done**. It has to be duplicated twice, one copy fires when a non-causal token is in **Memory** and the other copy when the token is causal. The **AccessGranted/AG** transition combines tokens from **BusIsFree** and **Pending** and moves a token into **Memory**. It thus has to be duplicated 4 times, once for each combination of causal/non-causal tokens in the source states. The same transformation is also applied to the **EC_{i,c}** transitions within the request generator, except when the transition is the source of the favorable variation.

Dropping Causality of Tokens: As discussed above, TPN causality does not distinguish why a token remains in a place. It might be due to an *intrinsic duration*, inherent to the activity the place represents, or *extrinsic blocking*, caused by contention on a shared resource (in our case, the bus). This distinction matters for temporal causality, since blocking introduces a delay that is not determined by the transition that produced the token, and therefore should not be treated as causal. We thus need a mechanism to drop temporal causality precisely in the cases where a token stays in a place due to blocking. We achieve this by applying a self-loop pattern on the relevant places: a transition that takes a causal token from the place and immediately returns the corresponding non-causal token, firing in zero time. This transition fires exactly when blocking is detected, effectively cutting the causal chain at that point while leaving the rest of the model’s behavior unchanged. In particular, since the request is blocked, the transition corresponding to granting access to the resource is not enabled, its clock is not running, so taking a concurrent transition to break causality does not affect this clock.

For instance, a token residing in **Memory** is making progress: it is waiting for the fixed memory access latency T_{mem} to elapse, which is an intrinsic duration associated with the activity itself, not an extrinsic blocking. Likewise, a token in a computation phase of the request generator is making progress even though it cannot yet move, since it is simply waiting for its computation time to elapse. In both cases, temporal causality is preserved, since the delay is entirely determined by the duration of the activity itself. This is not the case, however, for a token residing in **Pending**: there, the token is not making progress toward its own completion, but is instead blocked, waiting for an external condition – the bus becoming available and its turn being reached – that is unrelated to any intrinsic duration of its own.

This pattern occurs at exactly two places in our model, which both feed the transition **AccessGranted**. An access can only start when a request is pending and the bus is free, with the turn given to this request by the arbiter. The start of the access is therefore the latest of two events: the arrival of the request in **Pending**, and the release of the bus in **BusIsFree**. Only the latest of these two events determines the date of the access; the token that arrived first has to wait for the other one, and its delay is imposed from outside. Causality must thus be dropped for the token that waits. First, when a request arrives while the bus is used by another core, or while another core has priority, the request token is blocked in **Pending**: the date of its access is set by the end of the previous access, not by the core that issued the request. Second, when an access ends while no request is pending, the bus token is blocked in **BusIsFree**: the next access will start when the next request arrives, so its date no longer depends on the previous access.

The Transformed Model: If we apply this transformation to the original TPN model of Figure 1 from Section 3, we obtain the TPN depicted in Figure 4. Intuitively, a causal token in **BusIsFree** indicates that the previous bus grant was itself part of the causal chain of the favorable variation, while a causal token

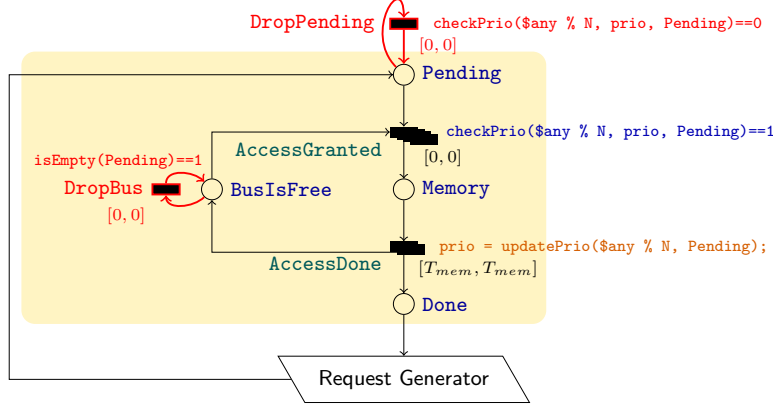


Fig. 4: RR arbiter () with a placeholder for a request generator ().

in **Pending** indicates that the corresponding request lies within that causal chain. Table 1 summarizes, for each duplicated transition, the resulting output color as a function of its input token colors. As shown in the table, the token produced in **Memory** is causal as soon as *either* the requesting token or the bus token is causal, and non-causal only when both inputs are non-causal. This reflects the fact that the request's admission to the bus at a given instant can be explained by the causal chain through either of its two determining factors: the request itself becoming eligible, or the bus becoming available. For **AccessDone**, the output color in **Done** simply preserves the color of the token consumed from **Memory**, since no additional resource contention occurs during the memory access itself.

The self-loop transitions on places **Pending** and **BusIsFree** ensure that causality is correctly broken. Usually a token could remain in these place, unable to progress, for as long as the guards of the transitions leaving these places do not evaluate to true. A token may only leave **Pending** when the function **checkPrio** returns true (1). Recall that the original model used N colors to represent the cores, whereas the transformed model uses $2 \cdot N$ colors. Since priority is defined at the granularity of cores, independently of whether a token is causal or

Table 1: Colored variants of **AccessGranted** and **AccessDone**.

Original	Input colors	Output color
AccessGranted	(Pending , BusIsFree) : (causal, causal)	causal
	(Pending , BusIsFree) : (causal, non-causal)	causal
	(Pending , BusIsFree) : (non-causal, causal)	causal
	(Pending , BusIsFree) : (non-causal, non-causal)	non-causal
AccessDone	Memory : causal	causal
	Memory : non-causal	non-causal

not, guards that reason about priority use the expression $\$any \% N$ to recover the originating core's identifier from the token color. **DropPending** fires immediately when a request/token is unable to acquire the bus, i.e., **checkPrio** returns false (0). The request is then not authorized to proceed and the corresponding token stays blocked in **Pending**, waiting for its turn. Similarly, **DropBus** fires immediately when no request seeks to acquire the bus, i.e., there is no token in **Pending** and the guard **isEmpty** returns true (1). Again, the token stays blocked in **BusIsFree**. In both cases, the tokens' stay in the place is not explained by any intrinsic duration, but by the absence of progress – exactly the condition under which temporal causality must be dropped. Other places in the model are not concerned, notably **Done** and **Memory**, since tokens cannot be blocked there due to the absence of guards in the transitions leaving them.

Note that the original model from Section 3 uses patterns frequently encountered in TPNs – notably the movement of tokens conceptually represents the progress of a request, a procedure, a protocol, or a computation, where guards often express blocking conditions. One might thus envision to automate the model transformation in order to add causality to a given TPN model: an operator, which takes as input the TPN (Figure 1 in our case), potential starting points (transitions corresponding to computations on cores in our case), and a set of dropping points (places and associated guards), that transforms the model automatically. The operator would modify transitions at starting points to produce tokens with causal colors, duplicate the other transitions accordingly to encode the propagation rules sketched above, and finally analyse the guards and firing intervals in order to insert self-loops to drop causality. The result would be an enhanced TPN, as illustrated by Figure 4, in which every token carries the additional causal/non-causal tag. More formally, we would have:

Definition 2. A *causality-enhancement* operator $\mathcal{T}_{ce}$ takes as input a TPN $N = (P, T, C, Var, pre, post, I_s, g, u, m_0)$ (cf. Definition 1), a start point given as a transition $t_0 \in T$ and a set D of dropping points, each given as a place $p \in P$ paired with a guard derived from g and pre of N . $\mathcal{T}_{ce}(N, t_0, D) = N'$ where N' is an output TPN, where each color of C is accompanied by a causal/non-causal boolean tag and subjected to three rules as follows:

- *creation* with t_0 consuming non-causal token and producing causal ones;
- *propagation* with every transition being duplicated to accommodate the causal/non-causal tokens at its input places, keeping unmodified I_s (firing interval), g (guard) and u (update), and the output being tagged causal if at least one input token is causal;
- *dropping* with a self-transition added to p , for each $(p, guard) \in D$, and guarded by its associated *guard* that turns a causal token into a non-causal one.

Observation 1. The causality-enhancement operator $\mathcal{T}_{ce}$ is *semantic-preserving* with respect to the initial system model since causal/non-causal tags are to be understood as annotations used to expose which transitions belong to the causal

chain. The creation and propagation rule keeps the original transitions unmodified with respect to firing interval, guards and the update operation. The dropping rule only adds zero-time self transitions that leave the underlying marking unchanged and that can be fired only when conflicting transitions are disabled thus not affecting their clock valuations.

Observation 2. The causality-enhancement operator $\mathcal{T}_{ce}$ adheres to the temporal causality definition introduced by Binder et al. [3] which states that a causal token stays in the causal chain (called causality-region in Binder et al. [3]) of the favorable variation at t_0 , i.e., the causal chain would allow tracing back to t_0 through a sequence of transition firings that contains none of the causality-dropping transitions added to the dropping points in D . This is exactly what dropping points are chosen to guarantee: each one captures whether tokens stay in their place due to blocking rather than the intrinsic duration of an activity so that a causal token is lost only when temporal causality itself would be broken.

Establishing the correctness claim of these observations would proceed by structural induction on the sequence of transition firings of $\mathcal{T}_{ce}(N, t_0, D)$, with the inductive step following the three rules in Definition 2. First, creation introduces the base case of the causal chain under construction, at t_0 . Second, propagation extends an on-going chain through a firing whenever at least one token carries a causal tag, and produces no chain otherwise. Third, dropping removes the causal tag exactly where the chain would be interrupted by blocking. Defining such an operator, formally proving its correctness, and investigating its applicability to relevant off-the-shelf TPN models is out of the scope of this work and left open as future work.

6 Investigating Counter-Intuitive Timing Anomalies

6.1 Counter-Intuitive Timing Anomalies under Round-Robin

In this section, we first illustrate our approach on a parameterized scenario under RR arbitration and show that it may exhibit a counter-intuitive TA under certain conditions. The scenario is comprised of two executions that are described in a single TPN model. One of the two executions experiences a favorable variation, where a core may have a shorter computation phase compared to the other reference execution. Starting from this variation, causality is tracked in the favorable case through our transformed model as defined in Section 5. The Roméo model checker is then able to construct a causal chain that leads to a relative slowdown at a later event, and whose execution verifies the three conditions of the formal definition of counter-intuitive TAs (cf. Section 2).

A Combined TPN Model: We consider a TPN consisting of two independent sub-models, α and β , each composed of an RR arbiter and request generator giving rise to two executions that execute side-by-side. Both executions are described by parameterized request generators shown in Figure 5. The parameters allow the

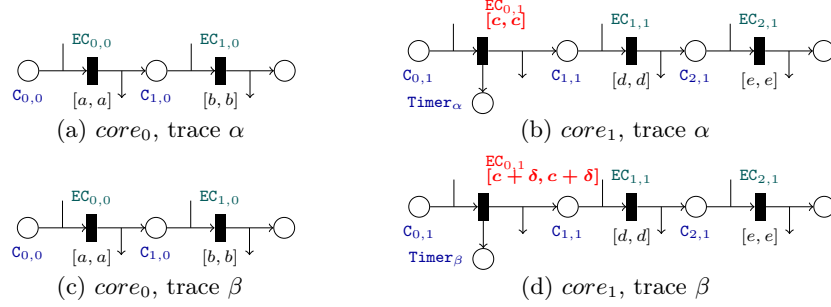


Fig. 5: Parameterized request generators, with a, b controlling the duration of the computation phases of $core_0$ and c, d, e as well as δ of $core_1$.

model checker to efficiently explore different execution variants that differ in the duration of the cores' computation phases expressed by a, b (for $core_0$) and c, d, e , as well as δ (for $core_1$). The parameters are identical between the two executions except for the computation time of the first request of $core_1$: in execution α the duration is c , while in execution β it is subject to a small additional delay δ , making it $c + \delta$. Execution α will thus experience the favorable variation and consequently might be subject to a potential TA. Sub-model α is hence transformed as described in Section 5, notably a causal token might be produced at the favorable variation. The other sub-model simply relies on the original TPN from Section 3. To compare the durations of the two executions, each request generator also contains a timer place, **Timer $_\alpha$** in α and **Timer $_\beta$** in β (Figure 5). The timer receives its token from **EC $_{0,1}$** , i.e., when the computation phases related to the variation end (at date c in α and $c + \delta$ in β), and is emptied when the execution terminates. **Timer $_\alpha$** is thus marked exactly during Δ_α , and **Timer $_\beta$** exactly during Δ_β . To compare the value of the two timers at the end of an execution, one can use the linear cost feature of Roméo, with a cost rate of 1 for each of the places **Timer $_\alpha$** and **Timer $_\beta$** .

Model Checking: As the duration of the computation phases are parametric, Roméo's parametric model checker can be used to explore the space of parameter valuations for which the property of interest holds. Rather than returning a single yes/no verdict for one fixed trace, the model checker returns a witness trace along with a set of constraints over the parameters – a combination of intervals and linear inequalities – that characterize exactly the parameter valuations under which the queried property is satisfied. Solving this system of constraints then can be used to derive a concrete trace that gives rise to the counter-intuitive TA. More precisely, we query the model checker for a trace in which $core_1$ experiences a favorable variation in execution α (*cf.*, the request generator) and producing a causal token that preserves causality until the end of the execution. The query is expressed in Roméo's temporal logic as:

$$\mathbf{EF} \left(\text{possibleTA}(\mathbf{FIN}_\alpha, \mathbf{Done}_\alpha, \mathbf{FIN}_\beta, \mathbf{Done}_\beta) = 1 \ \&\& \text{cost} < 0 \right).$$

We add two observer places, FIN_α and FIN_β , to executions α and β respectively: a token of colour i is produced in FIN_α (resp. FIN_β) when core_i issues its last request. A core has completed its sequence when, in addition, its last memory access has completed, i.e., when Done_α (resp. Done_β) contains its token. In execution α , Done_α also allows to track causality up to the completion of the last request. This allows us to tell whether core_i terminated with a causal token. The function *possibleTA* holds when all cores have completed their sequences, and when the final place Done_α contains a causal token, i.e., the end of execution α belongs to the causal region of the variation. The linear cost compares the two variations: places Timer_α and Timer_β are marked when the variation ends in α and β respectively, and the cost rate $\text{Timer}_\beta - \text{Timer}_\alpha$ accumulates $\Delta_\beta - \Delta_\alpha$. A negative cost thus characterizes a favorable variation in α . The relative slowdown $\Delta_\alpha > \Delta_\beta$ is then checked on the witness traces returned by the model checker.

The combined TPN, modeling executions α and β , together with the temporal logic formula of the above query, is submitted to the Roméo model checker, assuming $T_{\text{mem}} = 10$. We further constrain the model parameters so that the state space remains reasonable:

$$a \in [0, 2T_{\text{mem}}], \quad b \in [0, \infty[, \quad c \in [0, \infty[, \quad \delta \in]0, \infty[, \quad d = 0, \quad e = 0.$$

Leaving all parameters free leads to a state-space explosion, since each additional parameter multiplies the number of possible interleavings of the requests on the bus. We therefore keep some parameters free (b , c and δ), so that the model checker explores a large set of execution variants, while bounding or fixing the others (a , d and e), so that the analysis remains tractable. Roméo explores the parametric state space and returns the set of parameter valuations for which the query holds. This set is not a single interval: it is a union of 13 convex polyhedron, each described by intervals on the parameters and linear constraints

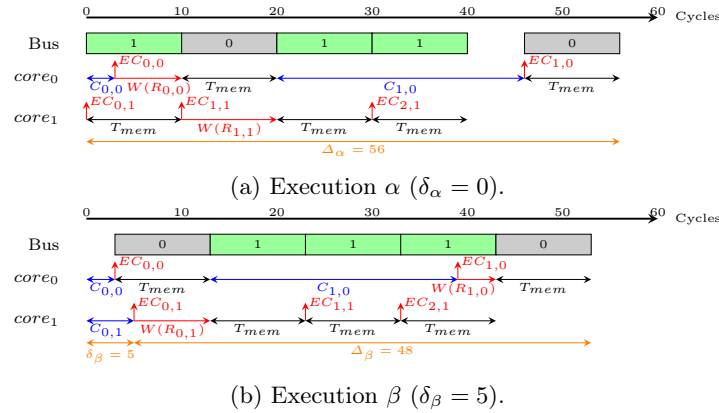


Fig. 6: A favorable variation in execution α ($\delta_\alpha = 0 < \delta_\beta = 5$) leading to a global slowdown ($\Delta_\alpha = 56 > \Delta_\beta = 48$), indicating a counter-intuitive TA.

between them. Each polyhedron corresponds to a different interleaving of the bus accesses of the two cores that leads to a TA. For readability, we only report one of them:

$$\begin{aligned} a &\in [0, 20], \quad b \in [20, 30], \quad c \in [0, 20], \quad \delta \in [1, 15], \\ a - c &\geq 0, \quad 1 \leq c + \delta - a \leq 10, \quad b + c - a \geq 20, \quad a + b - c \leq 30. \end{aligned}$$

Any valuation satisfying these constraints, as well as any valuation in the other 12 polyhedrons, gives rise to a counter-intuitive TA. The anomaly is therefore not an isolated example but holds over a larger part of the parameter domain. To illustrate it concretely, we select one such valuation and examine the resulting executions.

Figure 6 illustrates the two executions arising from the witness trace obtained for $a = 3, b = 26, c = 0, \delta = 5, d = 0$ and $e = 0$, which is a valuation that satisfies the constraints. Execution α completes in 56 time units, while β completes in 53 time units – which intuitively already appears to be a counter-intuitive TA due to the inverted completion order. In addition, the witness trace also satisfies the formal conditions:

Variation: As explained above, the first request of $core_1$ has a shorter computation time in trace α ($c = 0$) than in trace β ($c + \delta = 5$). This constitutes a favorable variation for α (cf. Section 2).

Slowdown: The slowdown is not expressed in terms of absolute completion times, but as the relative time distance from the transition that releases the favorable variation to a corresponding transition in both traces. The end of the variation in executions α and β is marked by the end of the first computation phase of $core_1$, which occurs at instant $c = 0$ and $c + \delta = 5$. Taking the completion of the last memory access as the point of comparison for the relative global slowdown, we obtain $\Delta_\alpha = 56 - 0 = 56$ and $\Delta_\beta = 53 - 5 = 48$. Since $\Delta_\alpha = 56 > \Delta_\beta = 48$, trace α exhibits a relative slowdown despite its favorable variation.

Causality: For this slowdown to qualify as a genuine counter-intuitive TA, it must be attributable to the favorable variation itself, rather than to some unrelated effect elsewhere in the trace. We verify this using the causal token created at the favorable variation: tracing its propagation through the transformed model, we observe that it remains present all the way to the place $Done_\alpha$ in the final state of trace α . The corresponding causal chain is illustrated by the red chain in Figure 3. This confirms that the observed slowdown lies within the causal region of the variation.

Since all three conditions – variation, slowdown, and causality – are satisfied, this witness trace confirms that the scenario constitutes a genuine counter-intuitive TA.

6.2 Counter-Intuitive Timing Anomalies under Time-Division Multiple Access

The approach presented in Section 5 is not limited to RR arbitration and can also be applied to other arbitration policy, such as Time-Division Multiple Access (TDMA) arbitration. The computation model of cores, the creation of

causality, and its propagation through the bus arbiter work in exactly the same way. The arbiter itself is however simplified as described next.

TDMA arbitration: Time is divided into slots of length T_{mem} , owned by the cores in a fixed cyclic order ($core_0, \dots, core_{N-1}, core_0, \dots$). We model this behavior as colored tokens (one for each core) moving in circles between two places (**SlotStart** and **Period**) as shown in Figure 7. At any moment in time only one token can be present in **SlotStart**, where the presence of a token of color c marks the beginning of the slot of $core_c$. The arbiter consequently only grants access to the memory when tokens of the same color are in **SlotStart** and **Pending** – this means that the arbiter is *not* work-conserving. Tokens immediately move from **SlotStart** to **Period**, where they spend the remaining time of the period ($N \cdot T_{mem}$), before returning to **SlotStart**. The first tokens for each core is supplied from dedicated places (**Start_i**) at time $i \cdot T_{mem}$.

The model is transformed according to the same rules as described in Section 5. The number of colors is doubled to represent causality and transitions are transformed in order to propagate causality.

For our experiment we use a simpler request generator, that emits only a single memory request for $core_1$. The arrival of this request is constrained by parameters a and b in executions α and β respectively, which are constrained as follows:

$$a \in \left[\frac{1}{2} \cdot T_{mem}, \frac{5}{2} T_{mem} \right], \quad b \in \left[2 \cdot T_{mem}, \frac{5}{2} T_{mem} \right], \quad a < b.$$

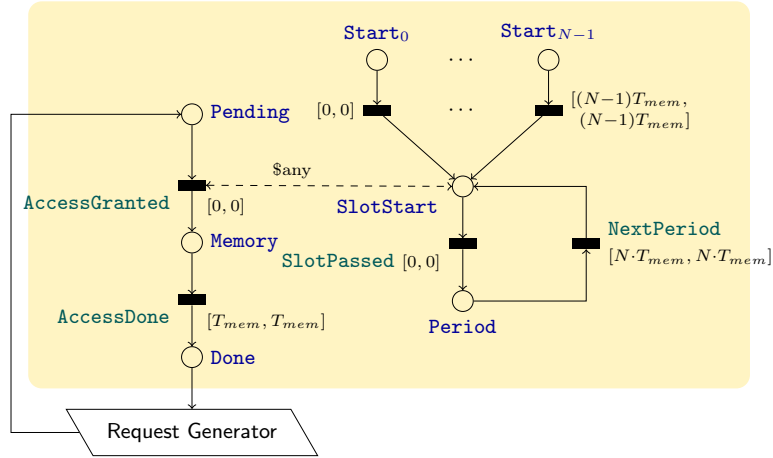


Fig. 7: TDMA arbiter (yellow box) with a placeholder for a request generator (trapezoid). Each core has one token, coloured by the core, which reaches **SlotStart** at the start of each slot of this core (offset $i \cdot T_{mem}$, period $N \cdot T_{mem}$) and leaves it at the same instant.

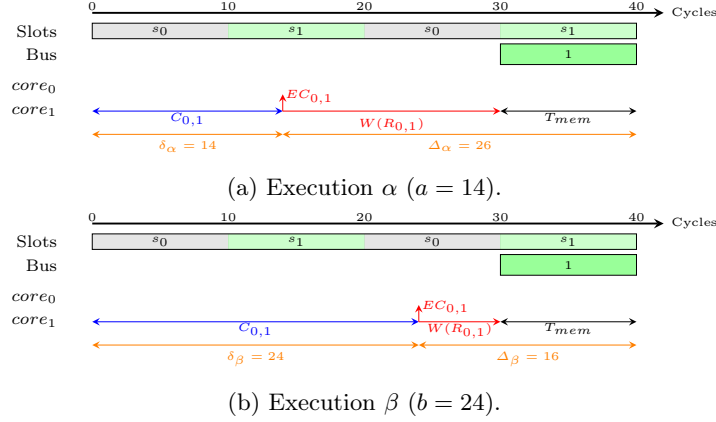


Fig. 8: TDMA arbitration: request of $core_1$ misses its slot in both executions and is granted access at $t = 30$. Both executions end at 40 and $\Delta_\alpha = 26 > \Delta_\beta = 16$.

We follow the same methodology as for the RR arbiter. The computation times are left as parameters, constrained as before. We then query the model checker for the existence of a causal counter-intuitive TA:

$$\mathbf{EF} \left(\text{possibleTA}(\mathbf{FIN}_\alpha, \mathbf{Done}_\alpha, \mathbf{FIN}_\beta, \mathbf{Done}_\beta) = 1 \ \&\& \text{cost} < 0 \right).$$

The query holds, and the model checker returns the set of parameter valuations for which such an anomaly exists: $a \in]T_{\text{mem}}, 2 \cdot T_{\text{mem}}]$, $b \in]2 \cdot T_{\text{mem}}, \frac{5}{2} \cdot T_{\text{mem}}]$, and $a < b$. In other words, an anomaly occurs whenever the request of $core_1$ is issued *after* the start of its first slot, whatever the other computation times. Notably this excludes possible valuations of a that would allow the request to be handled in the first slot, which would not result in a relative slowdown wrt. the variation. Also note that the relative slowdown of TAs is bounded by the TDM period $(N \cdot T_{\text{mem}} - 1)$, which explains why conventional upper bounds of the memory latency under TDM remain correct despite the presence of TAs.

Assuming $T_{\text{mem}} = 10$, a valid valuation is given by $a = 14$ and $b = 24$. In execution α , the first request of $core_1$ is issued at $t = 14$. It misses the slot starting at $t = 10$ and is granted access to the memory at the next slot of $core_1$, at $t = 30$. In execution β , the request is issued at $t = 24$ and is granted at the same slot. Both executions thus end at 40, resulting in $\Delta_\alpha = 26 > \Delta_\beta = 16$. The three conditions for a counter-intuitive TA are thus indeed reunited.

6.3 Discussion

Modeling choices: Each computation time is modeled by a point interval $[x, x]$, with parameter x being constrained by $x_{\min} \leq x \leq x_{\max}$. This is required for correctness: the two executions α and β must be tied, i.e., computation $C_{i,j}$ must have the same duration in both trace. Using parameters instead of fixed values

or intervals allows us to query Roméo for the values of the computation times for which a TA actually occurs.

Scalability: On our examples, the analysis takes 60 seconds for parameters with the ranges given in Section 6. The cost grows with the number of distinct behaviors allowed by the parameter ranges, as well as with the number of cores, and the length of the computation sequences. For instance, with the round-robin example, the model checker returns 13 zones, each one corresponding to a different arbitration order. Like all model checkers, Roméo is prone to state space explosion problems. However, we observe that our systems exhibit a set of favorable properties for exhaustive (symbolic) exploration: a limited degree of parallelism, boundedness in terms of both tokens and overall duration, and trivially bounded loops. Consequently, we believe it is indeed possible to study complex and interesting use cases. Note, however, that our main objective is to study the phenomenon of TAs and understand the mechanisms behind it in order to design counter-measures to prevent them or to safely over-approximate them. Roméo’s parameter synthesis is particularly useful for such a study as it allows to identify and compare the conditions causing TAs across a larger parameter space, instead of providing a single counter-example.

Application to real systems: Our goal is not to propose a detector capable of handling a complex scenario corresponding to the complete execution of a real system. Rather, we aim to conduct a detailed analysis of the interactions that arise in these systems and to better understand how they can lead to the occurrence of anomalies. We believe, however, that with our current framework, we will be able to study more realistic bus arbiter models and get closer to the protocols implemented in modern components. The mechanisms identified using Roméo can then be explored in more detailed models – such as cycle-accurate simulators – to determine whether they also pose a problem for real-world systems. This is a task we reserve for future work.

7 Related Work

Timing Anomalies: Lundqvist and Stenström [17] were the first to discover TAs in dynamically scheduled processors. They demonstrated that a cache miss may unexpectedly result in a shorter execution time than a cache hit. They referred to this counter-intuitive behavior as a counter-intuitive TA and also identified amplification TAs.

Wenzel et al. [24] explored the fundamental causes of timing anomalies in superscalar processors, accounting for the fact that different resource allocation decisions may be taken at runtime. Reineke et al. [20] were the first to attempt to formally define and classify TAs, distinguishes three classes: scheduling-, speculation-, and cache-related TAs. Contrary to Lundqvist and Stenström the phenomenon is not viewed as a hardware issue, but is instead related to abstractions in WCET analysis. This problem was later revisited [6,9], where the

authors argued that TAs should be defined directly on concrete hardware traces rather than using abstract analysis models. Cassez et al. [6], notably introduced the notion of *consistently slower hardware traces*, identifying TAs as properties of the underlying architecture rather than artifacts of abstraction. Gebhard [9] distinguished two types of self-reinforcing TAs called *domino effects*: K-bounded TAs stabilize after a bounded time whereas unbounded TAs effect program execution indefinitely. The work only provides a theoretical classification, without fundamentally characterizing domino effects. Our work might open possibilities in that direction. Domino effects may only manifest when causal tokens move on a cycle in the TPN. Checking the existence/properties of such cycles might yield a formal decision procedure. Binder et al. [4] evaluated of these and other existing definitions [20,7,6,9,14], showing that they are mutually inconsistent. Most attempts of providing a definition focused primarily on counter-intuitive TAs for processors or cache replacement policies [2]. These approaches all were theoretical and did not provide any implementation or tool support.

More recently, Binder et al. [3] revisited the definition of counter-intuitive TAs and argued that previous definitions lack an explicit notion of causality, relating local and global timing effects. Using the ETDG as an intermediate representation, they proposed a causality-based definition of TAs that was implemented with a decision procedure amenable to model checking in TLA+ [15]. Later the approach was extended to amplification TAs [21]. However, their formulation relies on the construction of the ETDG, whose architecture-specific rules are derived from expert knowledge in an ad-hoc manner. Their theoretic definition was reused as is in this work. However, our implementation relies on TPNs, which already have an inherent notion of TPN causality that, as demonstrated in Section 5 can be adapted to track Binder's notion of temporal causality. This work thus provides a more formal and solid foundation of the definition and its implementation in Roméo [8,16].

Shah et al. [22] explicitly investigated TAs, previously only considered at the processor core and cache level, in multi-core architectures. They identify new classes of TAs caused by interference on shared resources under a Round-Robin bus arbiter. The first anomaly shows that *more aggressive* interference from co-running applications may improve the execution time of the application under analysis, in short a counter-intuitive timing behavior. Unfortunately, the notion of aggressive interference is not formally defined. It appears to be based on merely counting the number of memory requests without taking their temporal relations into account. This allows to artificially inflate the number of *irrelevant* requests in order to influence the detection of a TA [5,19]. The second kind of anomaly is based on the notion of α -interference, which is supposed to maximize interference and represents α cores continuously performing memory accesses concurrently with a given application. Fewer interfering cores may induce longer execution times than a larger number of interfering cores. As stated by the authors themselves, α -interference does not necessarily lead to worst-case scenarios: "measured execution time in the presence of uninterrupted interference [is] a highly unreliable WCET candidate" [22]. It thus fails to satisfy the

author’s own objective of maximizing interference, questioning the suitability to characterize TAs [5,19]. In this work, a formal, causality-based approach is followed to systematically explore the timing behavior of multi-core systems at the bus arbitration level, using temporal causality to precisely characterize when and why a counter-intuitive TA arises.

Model Checking in Real-Time Systems: Model checking has been extensively used in the analysis of real-time systems. Next, without being exhaustive, we review representative model checking approaches for WCET analysis in the multi-core context, before turning to work that explicitly addresses TAs. Lv et al. [18] propose a fully integrated WCET analysis for multi-core applications by combining abstract interpretation with model checking. Abstract interpretation is first used to classify cache accesses (hit, miss, or unknown) at the basic block level. The resulting timing model is analyzed using UPPAAL [1] to derive WCET bounds. Similarly, Gustavsson et al. [10] employ timed automata to model both application control flow and shared hardware resources, relying on UPPAAL’s model checking engine to compute WCET bounds for multi-core architectures. This work is positioned as an early exploration of the feasibility of model checking for multi-core WCET analysis.

8 Conclusion

In this paper we are the first to investigate counter-intuitive TAs rigorously in multi-core systems by adopting a definition by Binder et al. [3] developed for out-of-order processor cores. The heart of our approach is to recast the *temporal causality* of that definition in terms of causality inherent to TPNs. Building on an existing TPN model of a Round-Robin arbiter, we introduced an approach to transform the model such that it captures temporal causality, replacing the ad-hoc construction rules underlying the ETDG of the previous work and thus providing a more formal and sound footing of the formalism. The proposed transformation was applied manually to the model, but appears to be amenable to mechanization by mostly applying simple structural modifications to the original TPN model. The transformation step that needs to ensure that temporal causality is correctly dropped appears to be the most challenging, requiring the analysis of guard conditions and firing intervals of transitions. The feasibility of our approach was demonstrated on a Round-Robin arbiter model, for which a genuine counter-intuitive TA was identified exploiting both the notion of temporal causality and the parameterization capabilities of the Roméo model checker. Future work includes exploring ways to automate this transformation and evaluating it on other TPN models, including processor cores, cache policies, and coherence mechanisms found in multi-cores. We also plan to investigate the nature and possible sources of domino effects by analyzing cycles in TPNs.

References

1. Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Softw. Pract. Exp.*, 41(2):133–142, 2011.
2. Christoph Berg. PLRU Cache Domino Effects. In Frank Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET’06)*, volume 4 of *Open Access Series in Informatics (OASICS)*, pages 1–3. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2006.
3. Benjamin Binder, Mihail Asavoaie, Florian Brandner, Belgacem Ben Hedia, and Mathieu Jan. The role of causality in a formal definition of timing anomalies. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 91–102, 2022.
4. Benjamin Binder, Mihail Asavoaie, Belgacem Ben Hedia, Florian Brandner, and Mathieu Jan. Is this still normal? putting definitions of timing anomalies to the test. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 139–148. IEEE, 2021.
5. Florian Brandner, Mihail Asavoaie, Jean-Luc Bechenec, Thomas Carle, Hugues Cassé, Sébastien Faucou, and Lionel Rieg. Deliverable D2.1: Study of Timing Anomalies Documented in the Literature. Technical report, VERIMAG UMR 5104, Université Grenoble Alpes, France, 2025.
6. Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. What is a Timing Anomaly? In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *Open Access Series in Informatics (OASICS)*, pages 1–12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.
7. 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 *International Symposium on Design and Diagnostics of Electronic Circuits and systems*, 2006.
8. Guillaume Gardey, Didier Lime, Morgan Magnin, and Olivier (H.) Roux. Romeo: A tool for analyzing time petri nets. In *Computer Aided Verification*, volume 3576, pages 418–423. Springer, 2005. Series Title: Lecture Notes in Computer Science.
9. Gernot Gebhard. Timing Anomalies Reloaded. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *Open Access Series in Informatics (OASICS)*, pages 1–10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.
10. Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *Open Access Series in Informatics (OASICS)*, pages 101–112. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.
11. Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36, 2015.
12. I. Haur, J.-L. Béchenec, and O. H. Roux. Model-checking of concurrent real-time software using high-level colored time petri nets with stopwatches. *Cybernetics and Systems*, 56(6), 2025.
13. Kurt Jensen. Coloured petri nets and the invariant-method. *Theoretical Computer Science*, 14(3):317–336, 1981.

14. Raimund Kirner, Albrecht Kadlec, and Peter Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *Euromicro Conference on Real-Time Systems*, pages 119–128, 2009.
15. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
16. Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A parametric model-checker for petri nets with stopwatches. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 54–57. Springer, 2009.
17. T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, pages 12–21, 1999.
18. Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Real-Time Systems Symposium*, pages 339–349, 2010.
19. Claire Maiza, Lionel Rieg, Mihail Asavoe, Jean-Luc Béchenec, Dominique Blouin, Florian Brandner, Thomas Carle, Hugues Cassé, Sébastien Faucou, Bruno Ferres, Pierre-Emmanuel Hladik, Jahier Erwan, Mathieu Jan, Éric Jenn, Loïc Jezequel, Dumitru Potop Butucaru, Isabelle Puaut, Pascal Raymond, Christine Rochange, Pascal Sotin, Catherine Parent-Vigouroux, Houssam Zahaf, Hector Chabot, Maha Essabyr, Louison Jeanmougin, and Hichem Rebhi. Collaborative Action on Timing InterferenCes: Summary and Perspectives at Mid-term. In *ERTS 2026 - Embedded Real-Time Systems*, pages 1–11, Toulouse, France, February 2026. ERTS.
20. Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In Frank Mueller, editor, *International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *Open Access Series in Informatics (OASICS)*, pages 1–6. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2006.
21. Lilia Rouizi, Mihail Asavoe, Benjamin Binder, Lionel Rieg, and Florian Brandner. Revisiting Timing Anomalies in Predictable In-Order Pipelines. In *Euromicro Conference on Real-Time Systems (ECRTS 2025)*, volume 335 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025.
22. Hardik Shah, Kai Huang, and Alois Knoll. Timing anomalies in multi-core architectures due to the interference on the shared resources. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 708–713, 2014.
23. Louis-Marie Traonouez, Didier Lime, and Olivier H. Roux. Parametric model-checking of time Petri nets with stopwatches using the state-class graph. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 5215 of *LNCS*, pages 280–294. Springer, 2008.
24. I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *International Conference on Quality Software (QSIC'05)*, pages 295–306. IEEE, 2005.