Master of Science in Informatics at Grenoble
Master Informatique
Specialization Parallel Computing and Distributed Systems

Timing Anomaly through Branch Prediction

Andrei Ilin

Defense Date, 2025

Research project performed at VERIMAG

Under the supervision of:
Lionel Rieg

Florian Brandner

Mihail Asavoae

June 2025

Abstract

In this report, we consider counter-intuitive timing anomalies (TAs) due to branch prediction in out-of-order (OoO) processor pipelines. A TA is a phenomenon when a local speedup in instruction execution leads to a global slowdown. We overview the existing TAs definitions and evaluate the applicability of the most promising one to branch prediction, using a novel ad-hoc model checking tool. Finally, we show that although the chosen definition adequately capture some TAs caused by branch prediction, its definition of causality cannot always capture the proper causality connections. This suggests that finding consistent TA definition encompassing branch prediction is still an open problem.

Résumé

Dans ce rapport, nous étudions les anomalies temporelles contre-intuitives dues à la prédiction de branchement dans les pipelines de processeurs à exécution dans le désordre. Une TA est un phénomène où une accélération locale de l'exécution d'une instruction conduit à un ralentissement global de l'exécution. Nous passons en revue les définitions existantes des TAs et évaluons l'applicabilité de la plus prometteuse à la prédiction de branchement, en utilisant un nouvel outil de model checking ad hoc. Enfin, nous montrons que, bien que la définition choisie capture correctement certaines TAs causées par la prédiction de branchement, sa définition de la causalité ne permet pas toujours de bien saisir les bons liens de causalité. Cela suggère que concevoir une définition cohérente des TAs englobant la prédiction de branchement reste un problème ouvert.

Contents

— 1 —
Introduction

TODO: **what is a critical system, how it is different**

In critical systems such as airplanes and cars, it is important that tasks running on the hardware meet their deadlines. For example, in a car's braking system, missing a program deadline can lead to loss of control over the vehicle. Therefore, it is crucial to have a rigorous analysis that can provide an upper bound for the program's execution time on a given hardware platform. In non-critical real-time applications, the execution time bound can be estimated by measuring the execution time for many input values. This gives the maximum observed execution time, which usually underestimates the real *worst-case execution time (WCET)*.

In critical real-time systems, this approach is not enough because some cases may be missed during observation, as shown in Figure 1.1. The real WCET can only be found by testing the program on all possible inputs, which is usually not possible due to the complexity of the state-space. Therefore, methods to estimate WCET bounds use abstractions of the program and hardware. These methods may overestimate the WCET because of simplifications, but they are more scalable.

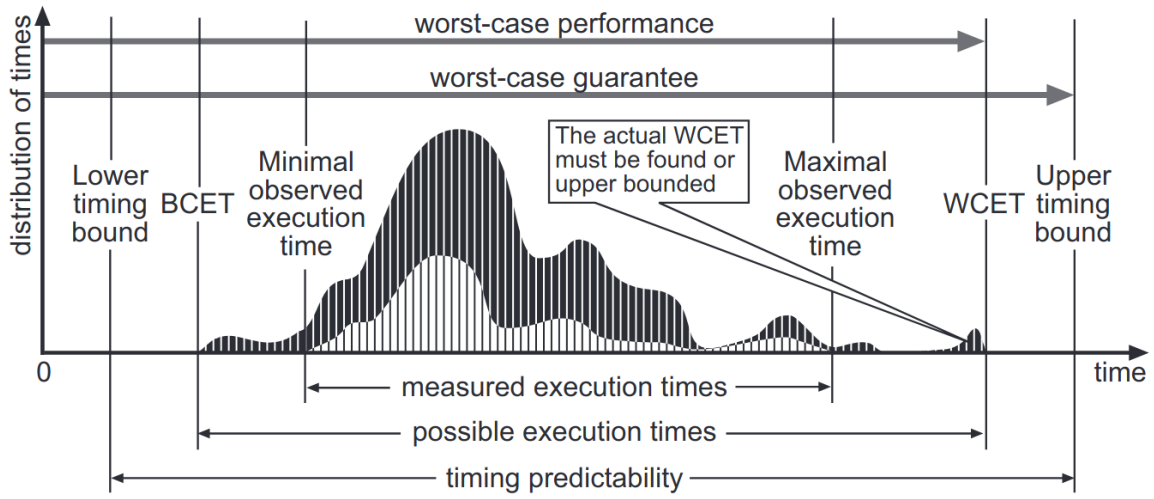


Figure 1.1: Timing analysis notations. The lower curve shows a subset of measured executions. The darker curve, an envelope of the former, shows the times of all executions (from [5]).

Usually, WCET analysis is divided into several phases, each focusing on a part of the hardware or software. For example, in the software part, memory analysis assigns address bounds

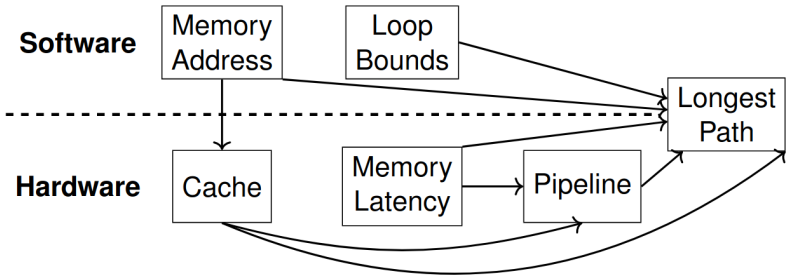


Figure 1.2: WCET analysis phases and their dependencies. (from [3])

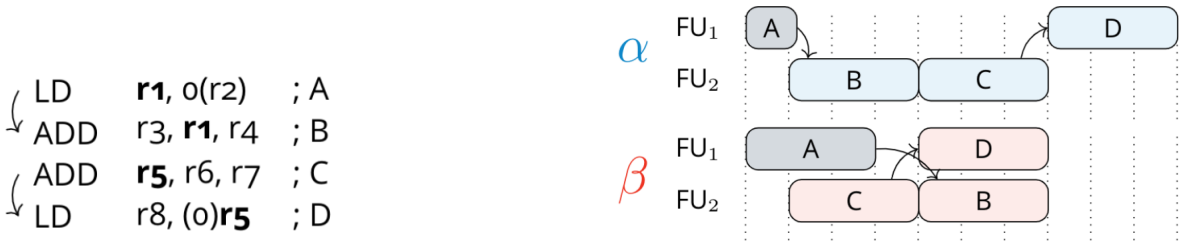
for each instruction [9], and loop bounds analysis finds the bounds for loop iterations as constants or formulas [10]. On the hardware side, there are analyzes for cache, main memory access latency, and pipeline. Together, these analyses help to find the longest path of a program, which gives the WCET bound.

Dividing WCET analysis into phases introduces the phase ordering problem: one phase may require information produced by another, leading to interdependencies. In some cases, these dependencies are circular. For instance, in Out-of-Order (OoO) processors, the instruction execution order depends on the architectural state, which also influences cache accesses. Consequently, cache analysis may depend on pipeline analysis. For timing analysis, composable systems are desirable [16].

Nevertheless, most architectures are not composable and exhibit so-called *timing anomalies (TAs)*. A TA arises when local worst-case scenarios do not necessarily lead to a global worst case. TAs are observed in pairs of execution traces with identical instruction sequences but differing initial hardware states. Variations in cache states, such as a miss in one trace and a hit in another, can result in timing differences. Example 1 illustrates how the initial cache state can induce a TA.

Example 1:

Figure 4.3 shows an example of such an anomaly. The assembly sequence has 4 instructions (A, B, C, D) with data dependencies $A \rightarrow B$ and $C \rightarrow D$. Figure 1.3b shows two traces (α, β) from running the program. There is a difference in the latency of instruction A (1 in α and 3 in β). In trace α, the variation seems better, but the total execution time is higher, which shows an anomaly.



(a) Input assembly sequence

(b) Scheduling on functional units

Figure 1.3: TA caused by variation in latency of instruction A (from [2])

Timing anomalies pose a significant challenge for timing predictability. Understanding their nature is essential for estimating their impact on global execution time and for designing hardware that avoids TAs. Various hardware features, such as caches, memory buses, and speculative execution, can introduce TAs. In real-time systems, some of these features are often disabled to improve predictability, which can result in reduced performance. One of the most critical optimizations is branch prediction; for example, Akkas et al. [1] demonstrated that this technique can increase performance by up to 37%. Therefore, it is important to study the timing behavior of branch prediction to leverage its performance benefits. In this work, we focus on TAs caused by branch predictors. We begin by explaining the microarchitectural context, then review existing definitions, and finally present our framework for identifying branch-related TAs and propose a formal definition for them.

— 2 —
Processor architecture

2.1 Instruction Set architecture

An Instruction Set Architecture (ISA) specifies the set of instructions and the registers they operate on. From the ISA perspective, all instructions are executed atomically, and their results become visible only after execution completes. The ISA serves as an interface between software and the underlying hardware microarchitecture that implements it. This abstraction enables software to reason about program behavior independently of the microarchitectural details, which may include additional internal registers or buffers and update state at a finer (cycle-level) granularity.

The ISA defines the binary format of instructions, which are stored in memory and accessed by the processor, typically through cache mechanisms. The processor operates in cycles, fetching instructions from memory and executing them. The *microarchitectural state* refers to the state of all hardware registers within the processor. Unlike the ISA, which defines state transitions at the instruction level, the microarchitectural state is defined at clock-cycle granularity, allowing instructions to take multiple cycles to complete.

2.2 Processor Pipeline Stages

Early processors executed instructions sequentially, completing one instruction before starting the next. This approach, known as non-pipelined or single-cycle execution, resulted in low throughput because each instruction had to wait for the previous one to finish all processing steps. Modern processors, in contrast, employ pipelining, which overlaps the execution of multiple instructions by dividing the process into discrete stages. This significantly improves instruction throughput and overall performance by utilizing hardware resources more efficiently.

Although the exact decomposition into stages may vary across architectures, a typical processor pipeline consists of five fundamental stages. More advanced processors may further subdivide these stages to enhance performance.

We start by describing the fundamental pipeline stages and then introduce some optimizations such as superscalar pipelines, out-of-order execution and branch prediction.

1. **Instruction Fetch (IF)**: The processor retrieves the next instruction from memory of the address from the program counter (PC) register. This access typically leverages the

instruction cache to reduce latency.

2. **Instruction Decode (ID)**: The fetched instruction, represented in binary format, is decoded to determine its type and the registers it operates on. During this stage, the required operands are read from the register file, and control signals are generated for subsequent pipeline stages depending on the instruction type.

3. **Execute (EX)**: The instruction is executed in this stage. Arithmetic and logic operations are performed by dedicated *functional units* (FUs), which are selected based on the decoded instruction type. For memory access or control flow instructions, the effective address or branch target is computed during this stage.

4. **Memory Access (MEM)**: If the instruction involves a memory operation (load or store), this stage accesses the memory hierarchy to read or write data. Non-memory instructions are not affected by this stage.

5. **Writeback (WB) or Commit (COM)**: The final stage writes the result of the instruction back to the register file, making it visible at the architectural (ISA) level. Only after this stage does the instruction's result become accessible to subsequent instructions.

2.3 Hazards

Although pipelining significantly improves throughput, it is still susceptible to stalls – situations where instruction progress is temporarily halted for one or more clock cycles. These stalls often result from dependencies and interactions among instructions within the program.

In this section, we provide an overview of these hazards and later we discuss hardware mechanisms designed to mitigate their impact.

2.3.1 Data Hazards

Pipeline stalls can arise due to dependencies between instructions that access the same registers. There are three primary types of register dependencies:

Read-After-Write (RAW) dependencies, also known as true data dependencies, occur when an instruction requires a value that is produced by a preceding instruction. For example, in the expression $(1+2*3)$, the addition depends on the result of the multiplication, creating a RAW dependency between the two operations.

Write-After-Write (WAW) dependencies occur when two instructions write to the same register. To preserve program correctness, the writes must occur in program order, ensuring that the final value in the register corresponds to the last write.

Write-After-Read (WAR) dependencies arise when a younger instruction writes to a register that a previous instruction still needs to read. If the write occurs before the read, the earlier instruction may read an incorrect value.

RAW hazards are inherent to all architectures and must be handled to ensure correct execution. WAW and WAR hazards do not occur in simple in-order pipelines but must be addressed in more complex architectures such as out-of-order model, which will will be discussed in more details in the subsequent sections.

2.3.2 Control Hazards

Control hazards, also known as branch hazards, occur when the pipeline cannot determine the address of the next instruction to fetch due to the presence of a branch instruction. The outcome of the branch is typically resolved in the execute stage, leaving the pipeline uncertain about which instruction to fetch next.

To address this uncertainty, the pipeline introduces bubbles – cycles in which no useful work is performed – until the branch outcome is known. Figure 2.1 illustrates a pipeline stall caused by a control hazard following a branch instruction.

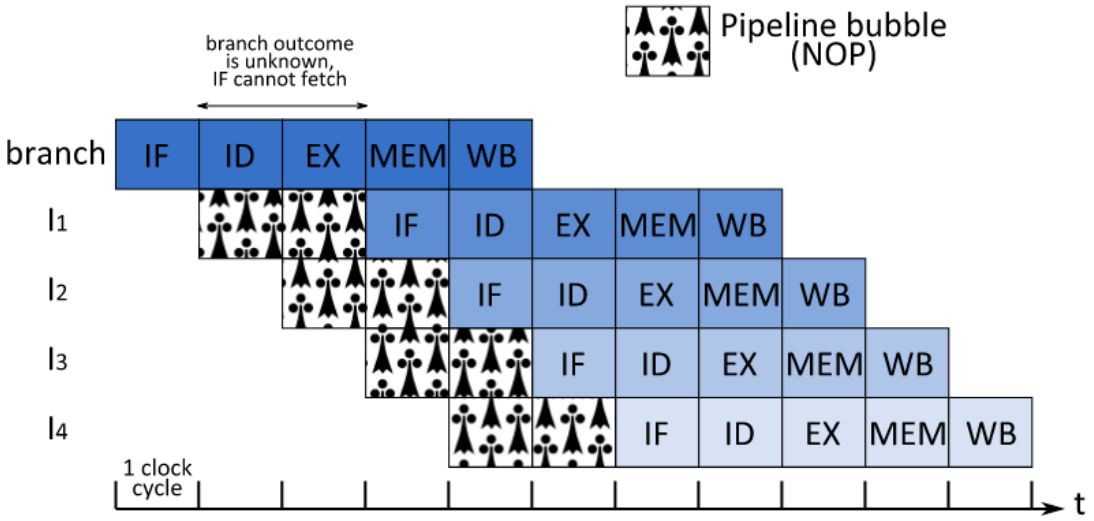


Figure 2.1: Example of control hazard: the pipeline is stalled until branch finished the execution (from [15])

2.4 Pipeline optimizations against data hazards

2.4.1 Bypass network

In a conventional pipeline, operands for an instruction are read from the register file during the ID stage. Consequently, if instruction *B* depends on the result of instruction *A*, *B* must wait until *A* completes its WB stage before it can proceed, introducing pipeline stalls. A bypass (or forwarding) network mitigates this delay by routing the result directly from the output of the EX or MEM stage to the input of the dependent instruction in the ID or EX stage, thereby reducing or eliminating the stall cycles caused by RAW dependencies. Figure 2.2 illustrates the effect of a bypass network on pipeline execution by comparing the 2 executions: without and with bypass network. As it can be observed, waiting for WB causes IF and ID stages to stall (noted with as *if* and *id* in lowercase).

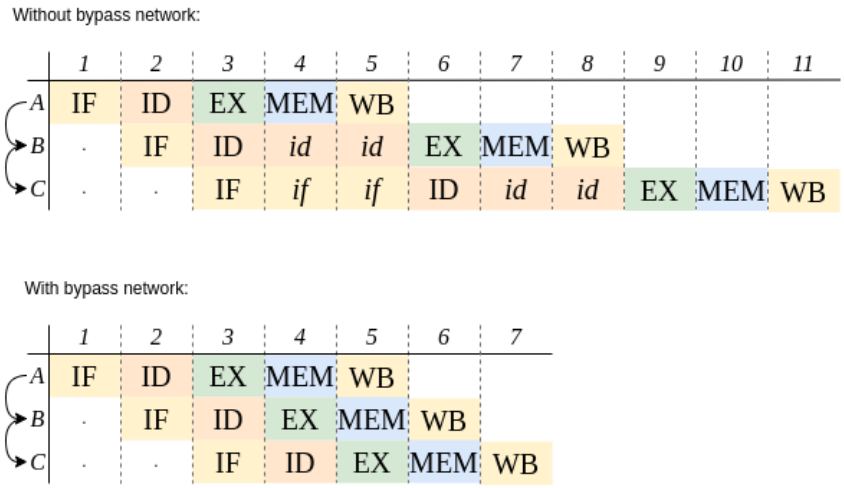


Figure 2.2: Execution of the same 3 instructions with RAW dependencies (noted with arrow). Comparison of pipeline without and with bypass network.

2.4.2 Superscalar Execution

Superscalar execution allows the processor to fetch and execute multiple instructions at the same time. This means that some pipeline stages are duplicated to handle several instructions in parallel. For example, in Figure 2.3, six instructions are fetched as three pairs, which saves one clock cycle for each pair compared to a non-superscalar pipeline. If the instructions are independent, this can greatly improve performance. However, duplicating every pipeline stage is expensive. While it is relatively easy to duplicate IF, ID, and COM stages, it is much harder to duplicate the execution and memory access stages, so creating high superscalar-degree pipelines comes with a great hardware cost.

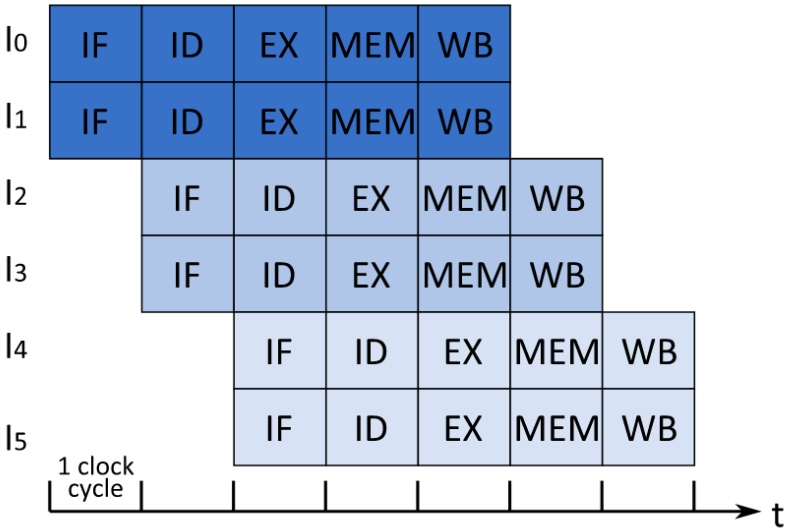


Figure 2.3: Degree 2 superscalar pipeline: each pipeline stage can process up to two instructions per cycle, provided the instructions are independent.

2.4.3 Out-of-Order (OoO) Pipeline

Out-of-Order (OoO) execution enables instructions to be processed based on data dependencies rather than strictly adhering to program order. While the architectural state (as defined by the ISA) must be updated in program order to preserve correctness, many instructions are independent and can be executed as soon as their operands are available. This parallelism improves resource utilization and overall throughput.

To support OoO execution while maintaining a consistent ISA-visible state, the processor pipeline is logically divided into in-order and out-of-order regions. The in-order region typically includes the instruction fetch (IF), decode (ID), and commit (COM) stages, while the out-of-order region encompasses execution (EX) and memory (MEM) access stages. Instructions are fetched and decoded in program order, but may be executed and completed out of order, provided their dependencies are satisfied. Final commitment to the architectural state occurs in order, ensuring correctness.

Key hardware structures that enable OoO execution:

Reservation Stations (RS): Each functional unit (FU) is associated with a reservation station, which buffers instructions waiting for execution. Once an instruction is decoded, it is dispatched to the appropriate RS based on its type. If the FU is busy or operands are not yet available, the instruction waits in the RS. The FU selects ready instructions from its RS for execution according to a scheduling policy. An example of such a policy could be that the instruction with a lower address is prioritized.

Reorder Buffer (ROB): The ROB is a FIFO structure that tracks all instructions in flight between decode and commit. When an instruction enters the out-of-order region, it is allocated an entry in the ROB. Upon completion of execution, the instruction is marked as complete in the ROB. The commit stage retires instructions from the ROB in program order, updating the architectural state only when instructions at the head of the ROB have finished execution. This mechanism ensures that the ISA state is updated in the correct order, even though execution may have occurred out of order.

Figure 2.4 shows the diagram of stage interconnections in processor. It represents the two optimizations discussed: superscalar and OoO execution.

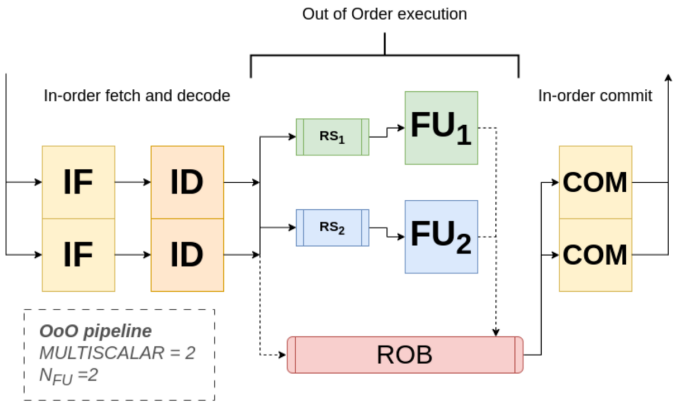


Figure 2.4: Out-of-Order superscalar pipeline diagram

2.5 Branch Prediction

Thus far, we have discussed optimizations that address issues arising from data dependencies. Another critical aspect is the control flow of the program. When encountering a control divergence, such as a conditional branch, the processor cannot determine which instruction to fetch next until the branch condition is resolved.

In the processor pipeline, the Instruction Fetch (IF) stage is responsible for retrieving the next instruction. However, when a conditional jump instruction is encountered, the address of the subsequent instruction remains undefined until the branch outcome is computed. As a result, the pipeline must stall until the branch instruction completes execution.

To improve efficiency, modern processors employ branch prediction mechanisms to guess the likely outcome of a branch and speculatively fetch the corresponding instruction. These speculatively executed instructions are not committed to the architectural state until the branch decision is confirmed. If the prediction is incorrect, the speculative instructions are flushed from the pipeline (this is also called *squashing*), and execution resumes from the correct path.

Branch prediction is by far one of the most efficient optimizations in modern processors. By accurately predicting the outcome of branches, processors can keep their pipelines filled and minimize the number of stalls caused by control hazards. This leads to significant improvements in instruction throughput and overall performance, especially in workloads with frequent branching. The effectiveness of branch prediction is a key factor in the performance of superscalar and deeply pipelined architectures. In the following sections we overview some well-known types of branch predictors.

2.5.1 Static Branch Predictors

Static branch prediction utilizes information available at compile time to determine the likely outcome of each branch. Several strategies are commonly employed:

Always Not Taken: This strategy assumes that branches are never taken, and the processor continues fetching instructions sequentially. It generally yields lower prediction accuracy, particularly in programs with frequent branching.

Always Taken: Here, the predictor assumes that every branch will be taken. This approach often achieves higher accuracy than the "always not taken" strategy, especially in code with loops, where branches are typically taken except at loop exit.

Backward Taken, Forward Not Taken (BTFNT): This method predicts that backward branches (those targeting a lower address) are taken, while forward branches are not taken. BTFNT is particularly effective for loop constructs, as loop-closing branches are usually backward and taken, whereas forward branches often correspond to loop exits or conditional statements.

While static branch prediction techniques are simple and require no additional hardware components, they suffer from several limitations. Their predictions are fixed and do not adapt to actual program behavior at runtime. As a result, they cannot exploit dynamic patterns or correlations in branch outcomes that may arise during execution. Static predictors also perform poorly in programs with complex or irregular control flow, where the direction of branches cannot be determined reliably at compile time.

2.5.2 Dynamic Branch Predictors

Dynamic Branch Predictors rely on information retrieved from execution and are usually based on previous branch outcomes. The usage of dynamic branch predictors requires additional hardware components which are discussed below.

Pattern History Table (PHT) is used to store information about each branch. It can be a bit denoting whether the branch was taken last time, or a more complex data. PHT is usually indexed by the lower bits of branch instruction address.

Branch Target Buffer (BTB) stores the destinations of previously computed branch. When starting speculative execution, values from BTB are used.

Return Stack Buffer (RSB) is used to predict the outcome of *ret* instructions.

One-Bit Predictor

The one-bit predictor is the simplest type of dynamic branch predictor. It uses PHT indexed by lower bits of address where one-bit value encodes the last branch outcome. Such a simple predictor is efficient when branch decision is not often changed throughout execution. For example, loop conditions are mispredicted only twice by this type of predictor: on the first and the last iterations of the loop.

However, more complex patterns diminish the efficiency of one-bit predictor. For instance, if the branch outcome changes each time, the predictor accuracy is zero.

Two-Bit Predictor

The two-bit predictor uses the same idea of PHT-indexing, but instead of storing just the outcome of previous branch, it has 4-state automaton encoded by 2 bits. The states are STRONG-TAKEN, WEAK-TAKEN, WEAK-NTAKEN and STRONG-NTAKEN. Figure 2.5 shows the transitions between the states.

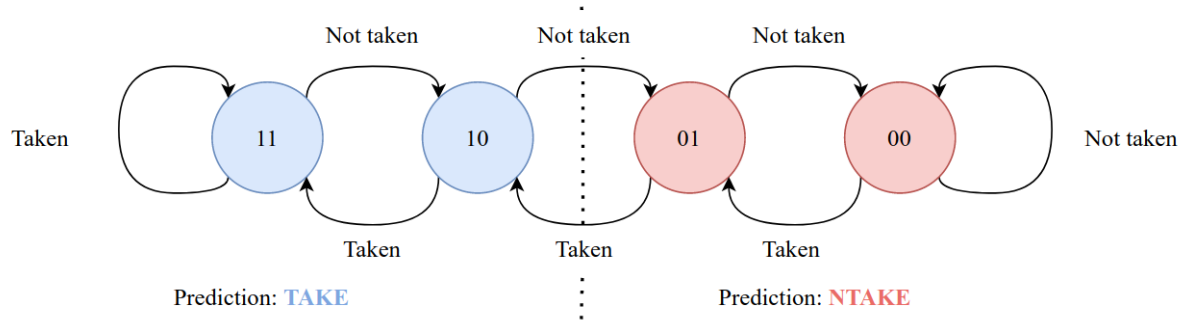


Figure 2.5: Two-bit predictor state machine (from [14])

The two-bit predictor outperforms the one-bit predictor because it requires two consecutive mispredictions before changing its prediction direction. This makes it more resilient to occasional anomalies, where a one-bit predictor would immediately flip its prediction after a single misprediction. As a result, the two-bit predictor achieves higher accuracy, especially in scenarios with repetitive branch behavior.

TODO: **Refer to comparison article**

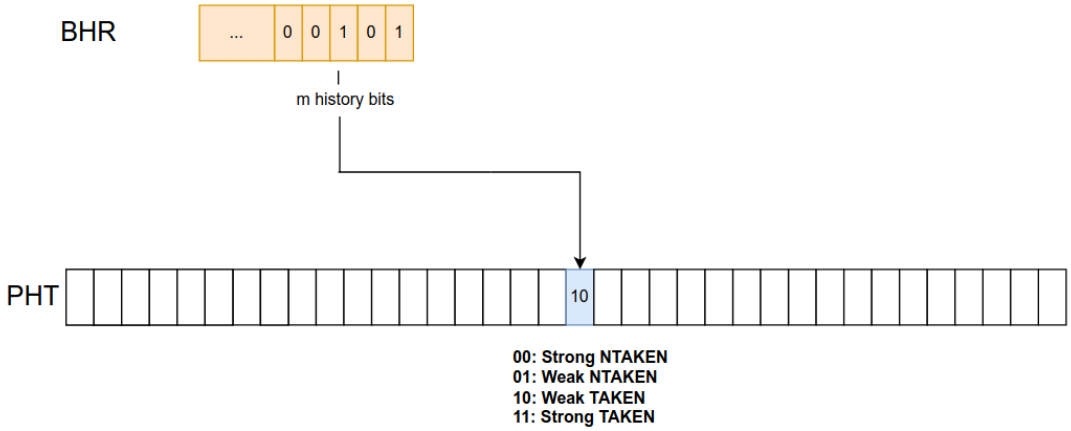


Figure 2.6: Two-Level Adaptive Training Predictor using a global BHR to index the PHT (from [14])

Two-Level Adaptive Training Predictor

In more advanced predictors the emphasis is on BHT-indexing. Two-bit predictor indexed by branch address+ may be good for simple structures like loops, but it fails to recognize more sophisticated patterns. *Two-Level Adaptive Training Predictor* addresses this issue by introducing a global *Branch History Register (BHR)* for each core. A BHR is a shift register storing the bits corresponding to outcomes of the last m branches. It is used to index BHT instead of branch address as shown in Figure 2.6. This type of predictor is capable of recognizing patterns induced by multiple branches.

The drawback of using the predictor we described is the use of a global history. For some branches the outcome is not related to the others, so an alternative version of adaptive predictor uses a *per-address branch history table (PBHT)* which stores multiple entries of history register which are indexed by the lower bitts of branch address (Figure 2.7).

Global Share Predictor

Both the global and local two-level adaptive predictors have their strengths: global history can capture correlations across branches, while local history can exploit patterns specific to individual branches. The *Global Share Predictor (gshare)* predictor combines these advantages in a simple and hardware-efficient way. It maintains a single global history recording the outcomes of the last m branches, as in the global predictor. However, instead of using the BHR directly to index the Pattern History Table (PHT), gshare computes the index by XOR-ing the BHR with n lower bits of the branch address (Figure 2.8). This hashing approach helps distinguish between different branches that may share similar global histories, reducing destructive aliasing in the PHT. XOR is chosen for its simplicity and low hardware cost, making gshare a practical and effective dynamic branch predictor.

2.5.3 Effect on Timing Analysis

As it is shown, there is a broad variety branch predictors. We have discussed only a few of them

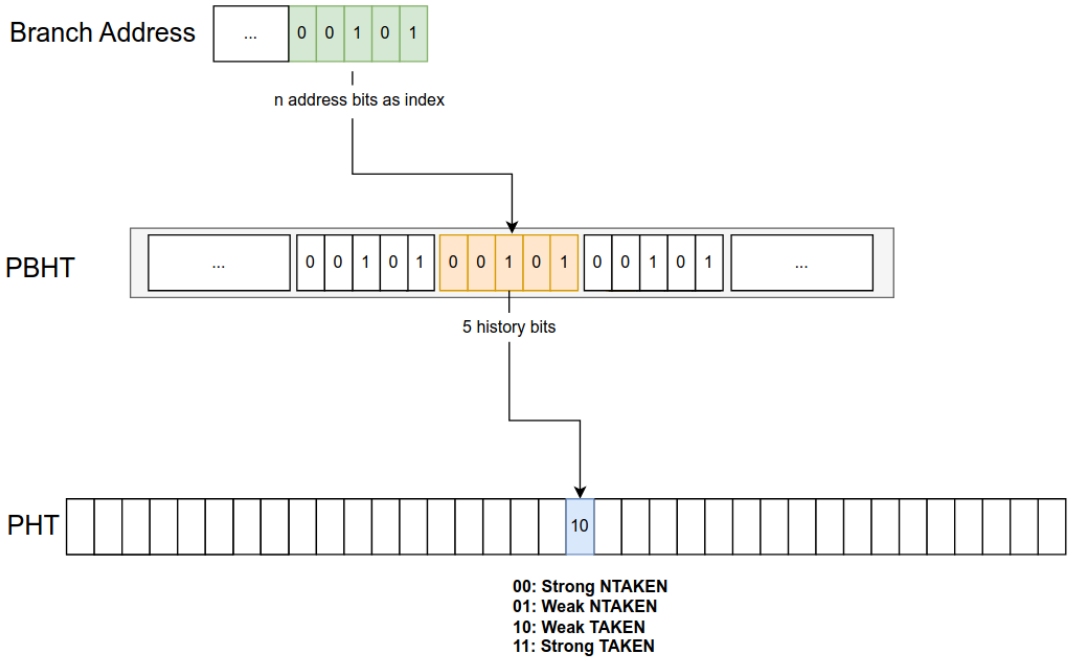


Figure 2.7: Two-Level Adaptive Training Predictor using a per-address BHR to index the PHT (from [14])

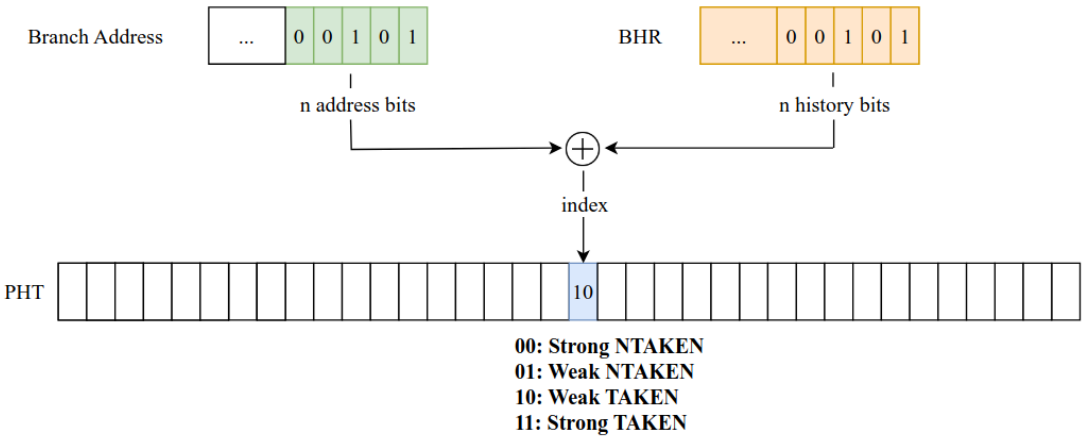


Figure 2.8: Global Share Predictor (from [14])

TODO: **!!!**

— 3 —
Timing Analysis and Anomalies

3.1 Evolution of TA-definitions

A *counterintuitive timing anomaly*, referred to as a *timing anomaly (TA)* in this work, is a phenomenon where a locally favorable condition results in a globally worse outcome (for example, a cache hit leading to a program slowdown). In contrast to counterintuitive anomalies, there also exist amplification timing anomalies, where the timing delay is greater than expected; however, this work focuses solely on counterintuitive anomalies. The concept of timing anomalies was first introduced by Lundqvist and Stenström in 1999 [13] in the context of timing analysis. TAs are architectural features that complicate the accurate analysis of timing behavior. Such anomalies can significantly impact execution time in ways not captured by worst-case execution time (WCET) analysis. Particularly problematic is the so-called domino effect, also described by Lundqvist and Stenström, which can lead to an unbounded slowdown due to the propagation of timing anomalies.

Despite the fact that timing anomalies have been known for a long time, the exact TA definition is a subject to debates. Since 1999 several attempts were made to formalize the notion of TA, some of them being more focused on the exact microarchitecture (like [7]) and some being more abstract and general (like [2], [8]). In this section we are giving an overview of existing definitions comparing their strengths and weaknesses.

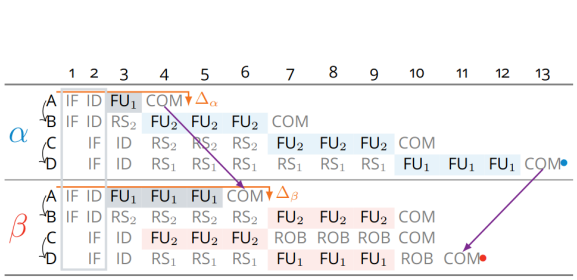
3.1.1 Step Heights

Gebhard [6] defines timing anomalies based on the comparison of local and global execution times. Specifically, a timing anomaly is present if an earlier instruction exhibits a shorter local execution time, while a later instruction in the same trace has a longer global execution time compared to another trace.

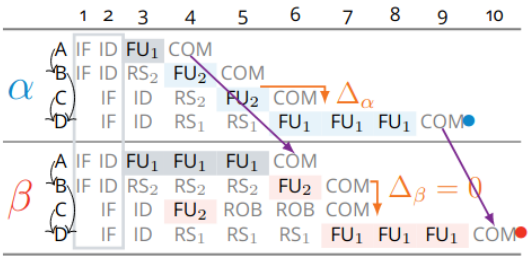
Figure 3.1a illustrates this definition using Example 1. The orange on the top arrow indicates the local execution time of instruction A. The global execution times for instruction D differ between traces α and β (13 and 11, respectively).

In his thesis [2], Binder provides a counterexample (Figure 3.1b), where it is clear that there is no TA (trace β has both unfavorable variation and longer execution time). However, Gebhard's definition signals an anomaly because of shorter local execution time of instruction C in trace β.

This poses a question whether it is reasonable to capture a local execution time as difference between instruction completion times.



(a) Interpretation of Example 1 using Gebhard's definition



(b) Counterexample to the definition

Figure 3.1: Gebhard's definition applied to execution traces (from [2])

3.1.2 Step-functions Intersections

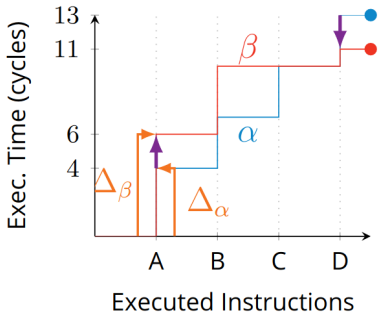


Figure 3.2: Execution time as step functions (from [2])

A similar definition is proposed by Cassez et al. [4]. The difference is that only the global execution time is taken into account. Thus, a TA arises when the step-functions (that map instructions to their absolute completion time) of two traces intersect. For example, Figure 3.2 shows the execution times as step functions from Example 1. The two functions intersect between C and D, thus signalling an anomaly.

This definition also leads to a misleading effect with a scenario found by Binder [2]. Figure 3.3 illustrates this by comparing two traces. The step-functions of traces α and β intersect, however there is no counter-intuitive TA happening as α is both longer and has a longer latency for the IF stage of instructions C and D.

3.1.3 Component Occupation

An alternative approach is proposed by Kirner et al. [11]. In their work the idea is to partition hardware into components and for each one define the occupation by instruction (for how many cycles it processes the instruction). A TA arises when a shorter component occupation coincides with a longer execution time in a chosen trace. For example, take Figure 3.1a and consider FU_1 and FU_2 as hardware components. In total, FU_1 is occupied for 4 cycles in trace α and 6 cycles in trace β. In the same time the execution time of α is longer, so a TA is identified.

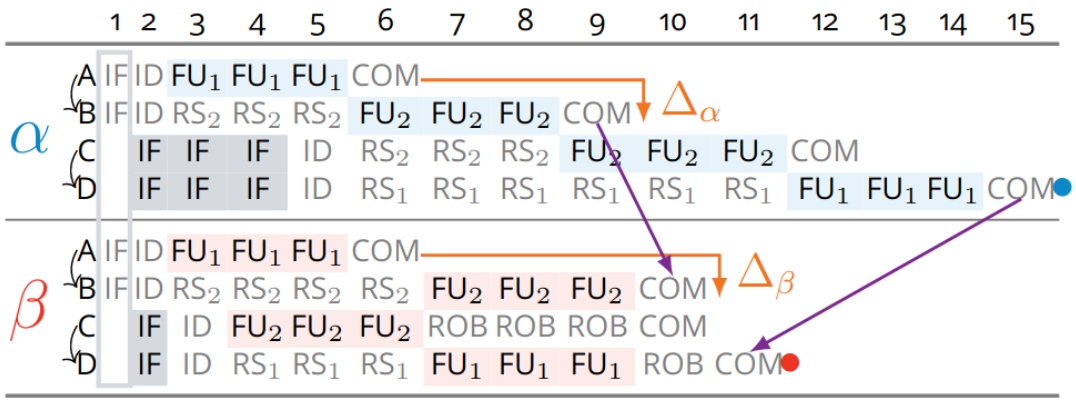


Figure 3.3: Contradicting result of Cassez's definition (from [2])

A main challenge with this definition is choosing the right components. The authors do not give a formal way to define what a component is or what properties it should have. For example, in the previous case, it is not clear if the chosen partitioning is valid, since functional units like FU_1 and FU_2 are controlled by the processor's instruction scheduling and thus are not fully independent. Another problem is that the occupation time of a component can change because of instructions that are not directly related to the timing anomaly. For instance, an earlier or later instruction, which does not affect the total execution time, might still reduce the occupation time of a component and cause a false positive. Similarly, false negatives can also occur. Additionally, if a resource switch happens (for example, an instruction is executed on a different functional unit in another trace), it can make the comparison of occupation times unreliable and further complicate the detection of timing anomalies.

TODO: **counterexample**

TODO: **Instruction Locality**

TODO: **Progress-based definition**

3.1.4 Event Time Dependency Graph

Binder et al. [2] define TAs using the notion of causality between events in an execution trace. In this work, a superscalar OoO pipeline is considered. The processor state is described as a composition of the states of each of the resources: *IF, ID, set of RS, set of FU, ROB, COM*. Each component holds information about the instruction it is currently processing, including required registers and remaining clock cycles.

The notion of an event is introduced based on qualitative changes in the pipeline associated with an instruction progressing through stages. An event from an execution trace (denoted as $e \in Events(\alpha)$) is a triple (i,r,t), where i is the instruction to which the event is related, r is the associated resource and the action (acquisition or release), and t is a timestamp corresponding to the clock cycle when the event occurs.

In the proposed framework, events correspond to the *IF, ID, FU*, and *COM* pipeline stages. For each instruction, there are seven types of events: $\uparrow$ IF, $\downarrow$ IF, $\uparrow$ ID, $\downarrow$ ID, $\uparrow$ FU, $\downarrow$ FU, and *COM*. The symbol $\uparrow$ denotes the acquisition of a resource, while $\downarrow$ indicates its release. *COM*

represents the acquisition of the commit stage; its release always occurs one clock cycle later, and since no subsequent stages exist, it is not further considered in the framework.

Latency is defined as the time difference between the acquisition and release of a resource. Each instruction passes through the same pipeline stages and is associated with the corresponding events. Thus, for each pair of traces corresponding to the same program, the sets of events differ only in their timestamps or, potentially, in the functional unit (FU) used, although resource switching is not modeled within this framework. Consequently, for each event in one trace, there exists a corresponding event in the other. Formally, this correspondence is defined by the function $CospEvent : Events(\alpha) \rightarrow Events(\beta)$.

Variation refers to the observation that the latency in one trace differs from the latency of the corresponding events in the other trace. For a pair of traces α and β, the variation is considered favorable for α if the latency in α is smaller than in β.

Variations are considered as the source of timing anomalies. They may represent different memory behaviors (e.g., cache hit or miss) for fetch and memory accesses in FU. Other sources of timing anomalies, such as memory bus contention or branching, are not considered in this framework.

The **Event Time Dependency Graph (ETDG)** of a trace τ, denoted as $G(\tau) = (\mathscr{N}, \mathscr{A})$, consists of a set of nodes $\mathscr{N} = Events(\tau)$ and a set of arcs $\mathscr{A} \subseteq \mathscr{N} \times \mathscr{N} \times \mathbb{N}$.

An arc is a triple (e_1, e_2, w) written as $e_1 \xrightarrow{w} e_2$, where e_1 is the source event node, e_2 is the destination node, and w is a lower bound of the delay between the two events. The arc means that at least w clock cycles must pass between e_1 and e_2.

Arcs are derived from a set of rules:

1. **Order of pipeline stages**

   Every instruction goes through the pipeline stages in a fixed order: first it is fetched, then decoded, then executed and only then committed. This creates a following dependencies for a given instruction I:

   - $(I, \downarrow \text{IF}, t_1) \xrightarrow{0} (I, \uparrow \text{ID}, t_2)$
   - $(I, \downarrow \text{ID}, t_3) \xrightarrow{0} (I, \uparrow \text{FU}, t_4)$
   - $(I, \downarrow \text{FU}, t_5) \xrightarrow{0} (I, COM, t_6)$

2. **Resource use**

   An instruction takes one or several clock cycles to go through each stage and cannot pass them faster.

   - $(I, \uparrow \text{IF}, t_0) \xrightarrow{lat_{IF}} (I, \downarrow \text{IF}, t_1)$
   - $(I, \uparrow \text{ID}, t_2) \xrightarrow{1} (I, \downarrow \text{ID}, t_3)$
   - $(I, \uparrow \text{FU}, t_4) \xrightarrow{lat_{FU}} (I, \downarrow \text{FU}, t_5)$

   lat_{IF} and lat_{FU} are the latencies of IF and FU stages respectively that come from how many clock cycles are required to fetch or execute a given instruction based on was it cache hit or miss for memory instructions.

   $lat_{IF} = t_1 - t_0$, $lat_{FU} = t_5 - t_4$

3. **Instruction order**

   The in-order part of the pipeline is constrained by instruction order. Thus, for successive instructions I_1 and I_2:

   $(I_1, \uparrow RES, t) \xrightarrow{0} (I_2, \uparrow RES, t'), RES \in \{IF, ID, COM\}$

4. **Data dependencies**

   A RAW dependency between I_1 and I_2 restricts the execution order of the instructions: $(I_1, \downarrow \mathrm{FU}, t) \xrightarrow{0} (I_2, \uparrow \mathrm{FU}, t')$.

5. **Resource contention**

   Also some instruction can be delayed because of limited resources. For instance, FU contention happens when I_1 and I_2 use the same FU, and it is busy by I_1 at the moment when I_2 is ready. This creates $(I_1, \downarrow \mathrm{FU}, t) \xrightarrow{0} (I_2, \uparrow \mathrm{FU}, t')$.

   Resource contention can also be caused by reaching the capacity limit of ROB or RS.

The **causality graph** is derived from the ETDG by removing unnecessary edges. For each event, we keep only the most relevant constraint. Only arcs of the form $e_1 \xrightarrow{e_2.time - e_1.time} e_2$ are left. Also, arcs related to variations are excluded, as in the case of a variation the delay of acquisition is induced by the hidden hardware state and not by the scheduling.

A **timing anomaly**, according to Binder, consists of three observations: variation, causality, and slowdown. If a trace exhibits a favorable variation and in its causal region there is an event that is delayed, then a TA is signaled.

Formally, a TA is observed on a pair of traces α and β if there exists a favorable variation in α relative to β. Let $\downarrow e_\alpha$ and $\downarrow e_\beta$ be the events corresponding to the end of the variation in both traces. If there exist events e'_α and e'_β, where $e'_\beta = CospEvent(e'_\alpha)$ and there is a path in the causality graph of α between $\downarrow e_\alpha$ and e'_α, such that $\Delta(\downarrow e_\beta, e'_\beta) < \Delta(\downarrow e_\alpha, e'_\alpha)$, where Δ is the delay between the two events, then a TA is signaled.

Figure 3.4 shows how the framework captures a TA for Example 1. Figure 3.5 presents the complete ETDG for trace α, with different dependency rules highlighted in different colors. The arcs reflecting causality are depicted with solid lines.

In contrast to other definitions, this one measures relative time from the acquisition of the resource instead of global time. This approach allows the separation of different variations and isolates the part of the trace that experiences the TA effect.

3.2 Conclusion

In this chapter, we reviewed the evolution of timing anomaly definitions, highlighting their strengths and limitations. We discussed approaches based on step heights, step-function intersections, component occupation, and causality graphs. While early definitions often led to misleading or ambiguous results, more recent formalizations – such as the event time dependency graph – offer a more precise and nuanced understanding of timing anomalies. Notably, Binder's event-based definition is the most promising in the context of branch prediction, as its flexible set of rules can be adapted to architectural details, enabling accurate modeling of

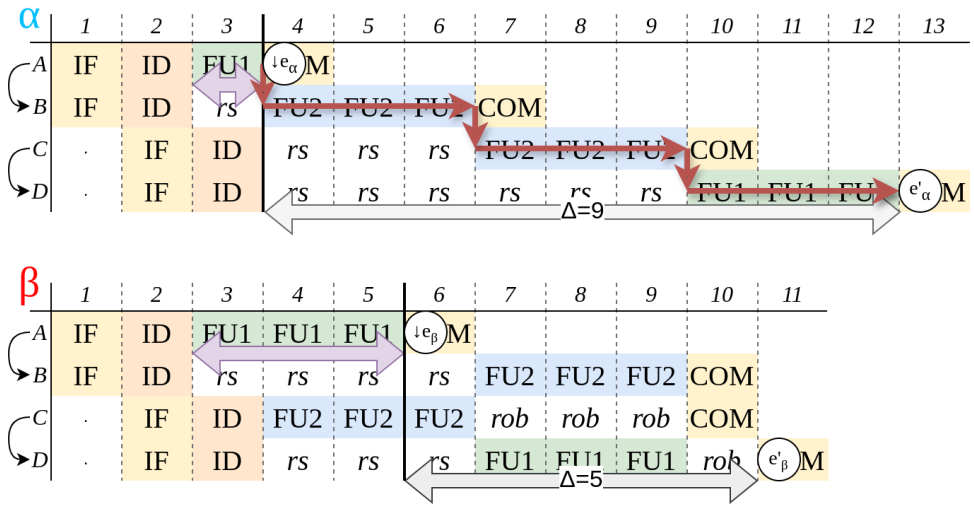


Figure 3.4: Causality-based TA detection applied to Example 1. $e_\alpha \downarrow = (A, \downarrow \text{FU}, 4), e_\beta \downarrow = (A, \downarrow \text{FU}, 6), e_\alpha = (A, COM, 13), e_\beta = (A, COM, 11)$. Purple arrow denotes latency which has a variation between two traces. Gray arrow shows delay between events which is greater in favorable trace. Causality in path α is marked by red arrows.

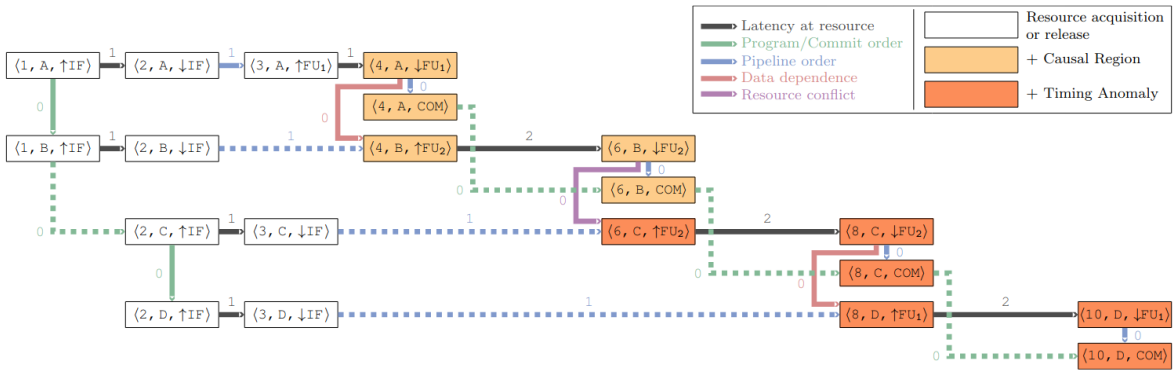


Figure 3.5: Complete ETDG for trace α from Figure 3.4, trace α. TODO: **image source**

complex behaviors. This progression underscores the complexity of accurately capturing timing anomalies and the importance of rigorous, architecture-aware definitions for reliable timing analysis.

+

— 4 —
Contribution

The definition by Binder et al. [2] provides a solid foundation for analyzing timing anomalies. However, it does not address branch prediction and related effects. Our goal is to extend this definition to cover such cases. We begin by reviewing Binder's framework and then introduce our own, which supports speculative execution. We describe a new input format that can represent speculative execution, present examples generated by our tool, and discuss how the existing definition can be adapted to these new scenarios.

4.1 Methodology

To systematically investigate timing anomalies induced by branch predictors, it is essential to efficiently generate and analyze relevant examples. Manual construction of such examples is both time-consuming and error-prone, motivating the need for a tool that can automatically or semi-automatically produce and validate them. With such a tool, we can iteratively generate candidate scenarios, analyze their behavior, and assess the applicability of Binder et al.'s definition to these new cases. This process enables us to refine and adapt the definition as necessary, guided by empirical evidence from the generated examples.

Our initial efforts focused on studying and extending the TLA$^+$ [12] framework developed by Binder et al. to support branch behavior. However, we encountered significant performance limitations and found the input format insufficiently flexible for rapid prototyping and adjustment of examples.

Consequently, we reimplemented the framework in C++, drawing on insights from the TLA$^+$ model. We also use TLA$^+$ as a reference to check the correctness. The new implementation offers substantial performance improvements and introduces randomized search capabilities, enabling efficient exploration of the space of possible instruction traces and facilitating the discovery of timing anomalies related to branch prediction.

4.2 Framework

4.2.1 Existing Framework Overview

Exploration by Model Checking

The implementation provided by Binder is written in TLA$^+$ [12]. The pipeline state is specified in set-theory notation. One model checker step corresponds to one clock cycle and derives a new HW state from the previous one. This allows simulation of non-deterministic timing behavior: each time a variation can happen, multiple next states are generated. TLA$^+$ covers all reachable states, ensuring that all possible behaviors are covered. * A pair of traces constitutes a whole model state. A TA is expressed as an invariant on pair of traces, so it is verified at each model checking step.

As well as construction of traces, the framework provides visualization methods for the traces and ETDG.

Input Trace Format

The input of the framework is a pair of:

1. Pipeline parameters: the superscalar degree, FU latencies, and memory access latencies depending on the cache events (hit or miss).

2. Instruction sequence: for each instruction, its type and registers are specified, as well as the set of cache behaviors to be explored by the model checker. The type is used to determine which FU will be used by the instruction, and the registers are used to derive the data dependencies.

Figure 4.1 illustrates the instruction sequence that causes a TA in Example 1. We can simplify this view by directly expressing the resource, dependencies, and possible latencies of each instruction. Figure 4.2 shows the input for the instruction trace from Example 1. The first column is the instruction label, the second is the resource used, the third is the set of data dependencies, and the last one captures possible execution latencies. In the same fashion, we could specify variations of latencies for the IF stage, but we skip them for simplicity. This table is sufficient to express a pair of execution traces derived from the instruction trace.

```
missLat == 3
mayDMiss == {1}
program == <<
[ ind |-> 1, type |-> "MemRead",  r0 |-> "ra", r1 |-> "",   r2 |-> "", addr |-> "0x1" ],
[ ind |-> 2, type |-> "IntAlu",   r0 |-> "",   r1 |-> "ra", r2 |-> "", addr |-> "0x2" ],
[ ind |-> 3, type |-> "IntAlu",   r0 |-> "rb", r1 |-> "",   r2 |-> "", addr |-> "0x3" ],
[ ind |-> 4, type |-> "MemWrite", r0 |-> "",   r1 |-> "rb", r2 |-> "", addr |-> "0x4" ]
>>
```

Figure 4.1: TLA$^+$ input format for Binder's framework. Some lines are excluded for brevity

	Resource	Dependencies	Latencies
A:	FU1		1|3
B:	FU2	$\{A\}$	3
C:	FU2		3
D:	FU1	$\{C\}$	3

Figure 4.2: Simplified input format of example from Figure 1.3a

4.2.2 Limitations

Despite using a model checker, the existing framework is capable of exploring only the traces that fit the instruction template. This limits the explored space to what is manually defined by the user. Also, there is no way to express branches and speculative execution.

Nevertheless, the framework may be used to manually specify the instruction trace using a template and generate the resulting pair of execution traces. This allows one to quickly sketch examples and analyze them. Unfortunately, this feature comes with some issues.

The most significant flaw we noticed was the performance. Firstly, TLA$^+$ itself takes a few seconds to generate the initial states of the model. Secondly, the graph is analyzed using a Java embedding, which calls a script in Python that in turn deserializes a graph from the text output of the model checker tool.

Moreover, the input is specified in lengthy TLA$^+$ notation, which prevents fast sketching of examples. Thus, it was decided not to write an extension of the existing framework, but to design a new one from scratch.

4.2.3 Our Novel Framework

We introduce a novel framework inspired by Binder et al., designed to address the limitations of the original implementation. Our framework features a lightweight input format that natively supports branch behavior and speculative execution, enabling concise and intuitive specification of instruction traces. To overcome the performance bottlenecks of TLA$^+$, we implement our solution in C++, providing significant speedup and enabling real-time feedback for rapid prototyping. Also, the performance enhancement allows to explore the larger state spaces effectively. Our framework facilitates efficient analysis of timing anomalies and supports both manual and automated exploration modes.

Misprediction Region

The format of the input traces was adapted to handle speculative execution. We decided to use a simplified format as in Figure 4.2 as a baseline. In Binder's framework, the instruction trace format is straightforward: it specifies all instructions that are fetched, executed, and finally committed. In the case of speculative execution, some instructions enter the pipeline but are never committed, being squashed by the resolution of the branch. To tackle this problem, we introduce the notion of the *misprediction region of a branch instruction*.

As an input trace, we specify all instructions that can enter the processor pipeline. As we focus only on the timing behavior of the program, abstracting from memory and register state, we also assume that the control flow is known for a given instruction trace. Thus, for each branch, we may specify the instructions in only one branch in the case of correct prediction. However,

in the case of misprediction, the instructions from the incorrect branch are fetched until the branch is resolved. We call such instructions *mispredicted*, and the set of such instructions after the branch a *misprediction region*.

In our input format, each line describes a single instruction, beginning with the functional unit to be used (`FU1`, `FU2`, etc.). This may be followed by an optional label, prefixed with `#`. Data dependencies can be specified by listing the labels of dependent instructions, each prefixed with `@`. Next, the possible execution latencies are provided as a list. For branch instructions, an optional `*` denotes variation in branch prediction behavior. Misprediction regions are indicated by indentation: an indented instruction belongs to the misprediction region of the most recent less-indented branch instruction.

Figures 4.3a and 4.3b present an example of the input format. Figure 4.3a displays the raw input as understood by the framework, while Figure 4.3b provides a more readable, tabular representation that will be used throughout the remainder of this article. In this example, instructions *C* and *D* reside within the misprediction region of instruction *B*. Figure 4.4 illustrates the two possible execution traces derived from this instruction trace: trace α corresponds to correct branch prediction, where *C* and *D* are skipped and never enter the pipeline; trace β demonstrates the misprediction scenario, in which *C* and *D* are fetched but subsequently squashed from the pipeline at clock cycle 5.

```
1 FU1   #1  [4]
2 FU2   @1  [1]  *
3     FU2   [4]
4     FU2   [4]
5 FU2       [4]
```

(a) Input format understandable for framework

		Res	Dep.	Lat.
A:		FU1		4
**B:*		FU2	{*A*}	1
	C:	FU2		4
	D:	FU2		4
E:		FU2		4

(b) Input format used further in the text

Figure 4.3: Two equivalent representations of input format supporting speculative execution

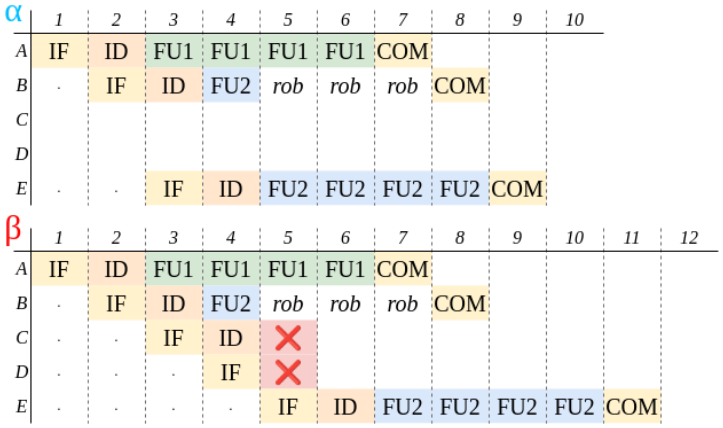


Figure 4.4: Pair of traces with correct and incorrect predictions. The squashing event is denoted with a red cross.

TODO: **nested mispred region**
TODO: **,ispred region should be sufficiently large**

Framework Implementation

We decided to use C++ [17] as the implementation language, as it is fast and includes a number of useful data structures in the standard library.

We define a single instruction as follows. It consists of the type of FU to be scheduled at (we do not consider resource switching), the latency in this FU, and a set of RAW dependencies. If the instruction is a branch, `mispred_region` is set to a positive n, denoting that the next n instructions are in the misprediction region of the current instruction. In case of no-branch instructions `mispred_region` is set to 0. If we want to model only the correct prediction, then `mispred_region` is set to 0; in this way, the branch behaves as an ordinary instruction. The `br_pred` flag specifies if the prediction is correct, which is needed when generating a pair of traces with variation in branch behavior.

```
struct Instr {
    int              fu_type = 0;
    int              lat_fu = 1;
    std::set<int>    data_deps;
    int              mispred_region = 0;
    bool             br_pred = false;
};
```

At the core of our framework is the `PipelineState` structure, which models the state of all pipeline stages. The `executed` set tracks instructions that have completed execution in the functional units, enabling dependency resolution. The `branch_stack` maintains the context for misprediction regions: each time a branch is fetched, it is pushed onto the stack and remains there until resolved. Together with the `squashed` set, this mechanism ensures correct handling of mispredicted regions. For simplicity, we do not impose capacity limits on the reservation stations (RS) or reorder buffer (ROB). The `next()` function advances the pipeline state by one clock cycle and returns whether execution has completed. It operates on the instruction sequence, which is accessed via the program counter (`pc`).

To obtain an execution trace from a given instruction sequence, we initialize an empty pipeline state (with no instructions present) and repeatedly call `next()` until the final state is reached. This process yields a sequence of pipeline states, which together form an execution trace.

```
struct StageEntry {
    int idx = -1;
    int cycles_left = 0;
};

struct PipelineState {
    int clock_cycle = 0;
    int pc = 0;
    vector<StageEntry>  stage_IF =  vector<StageEntry>(SUPERSCALAR);
    vector<int>         stage_ID =  vector<int>(SUPERSCALAR);
    vector<set<int>>    stage_RS =  vector<set<int>>(FU_NUM);
    vector<StageEntry>  stage_FU =  vector<StageEntry>(FU_NUM);
    vector<int>         stage_COM = vector<int>(SUPERSCALAR);
    deque<int>          ROB =       deque<int>();
    set<int>    executed;
    set<int>    squashed;
    vector<int> branch_stack;

```

```
19      bool next(const vector<Instr>& prog);
20  };
```

To enable efficient exploration of trace pairs that demonstrate timing anomalies (TA) and to support analysis over larger state spaces, not limited to a fixed instruction trace template, the framework provides three operating modes:

1. **Manual mode**: The user provides an instruction trace in the format given above. The framework then generates the corresponding pair of execution traces. This mode enables rapid construction and analysis of custom scenarios.

2. **Random search**: The framework generates random instruction traces within user-defined constraints and checks the resulting execution traces against a specified property. For example, it can explore all traces of length 5 containing one branch instruction and at most two RAW dependencies. While this method cannot guarantee exhaustive coverage of the state space, it is effective for quickly finding counterexamples in large spaces.

3. **State exploration**: The trace template is specified as a generator function, similar to random search mode. The framework then exhaustively verifies the property on every possible input, ensuring complete state space coverage. This mode is useful for proving properties about the model, but may be inefficient for finding counterexamples in large spaces due to the potential for excessive exploration of uninteresting subspaces.

In summary, we created a tool capable of studying traces both in automated and guided way. Our time-efficient implementation enables exploration of significantly larger state spaces that are infeasible to analyze using Binder's original framework. While TLA$^+$ offers greater expressive power for formalizing properties such as leveraging temporal logic, in the context of Binder et al., the verified property was ultimately specified as a state predicate embedded in Python code. Therefore, we believe that our choice of implementation does not result in a substantial loss of expressiveness or rigor for the intended analyses.

4.3 Generating TA Examples

We begin by generating representative examples of branch-induced timing anomalies (TAs). Our working hypothesis is that correct branch prediction can, in certain scenarios, result in longer execution times compared to cases with misprediction. To obtain minimal and illustrative examples, we perform an exhaustive search over the space of programs characterized by the following constraints:

1. 4 committed instructions with;

2. At most 2 dependencies;

3. 1 branch instruction.

The latency of the branch instruction was set to 1, reflecting the fact that conditional jumps typically involve simple, single-cycle operations (e.g., equality, greater-than-or-equal, less-than-or-equal comparisons). The Latency of all other instructions in this setting was set to 4. We considered a single-issue pipeline equipped with two functional units. Our framework

explored 4608 input traces in approximately 150 ms, identifying two timing anomalies, as detailed in Examples 2 and 3.

Example 2:

Figure 4.5a presents an instruction trace identified through random example generation. Here, C is a branch instruction with a misprediction region comprising $\{D, E\}$. The only data dependency is $A \rightarrow B$. The corresponding pair of execution traces is depicted in Figure 4.6a. In trace α, instructions D and E are skipped due to correct branch prediction, allowing F to be fetched immediately. This results in F executing earlier and occupying $FU2$ at clock cycle 7, which in turn delays the execution of instruction B and leads to an overall slowdown.

Example 3:

The second timing anomaly, illustrated in Figures 4.5b and 4.6b, differs from Example 2 only in the functional unit assigned to instruction C; all other instructions remain unchanged. Additionally, the misprediction region is extended due to the increased delay between branch prediction and branch resolution.

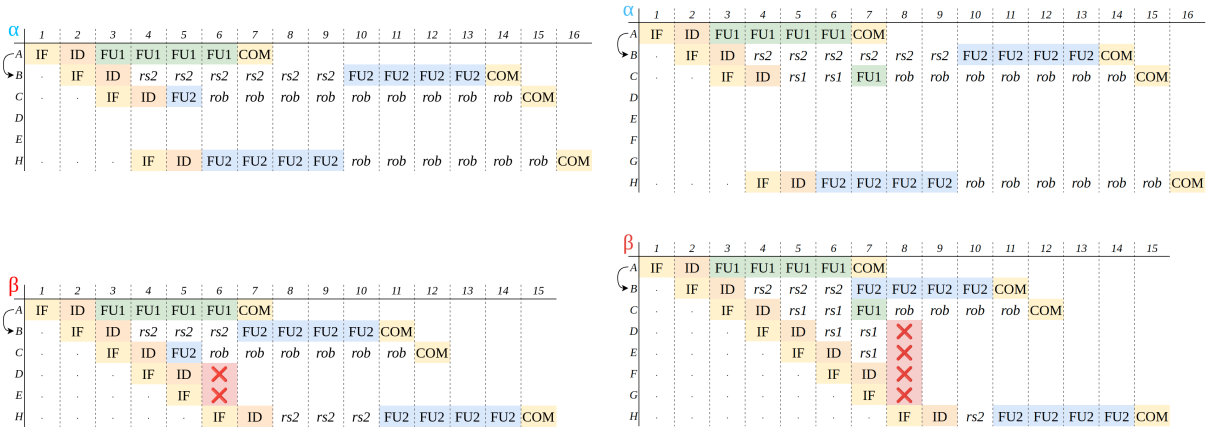
	Res.	Dep.	Lat.
A:	FU1		4
B:	FU2	$\{A\}$	4
***C:**	**FU2**		1
D:	FU1		4
E:	FU1		4
H:	FU2		4

(a) Input from Example 2

	Res.	Dep.	Lat.
A:	FU1		4
B:	FU2	$\{A\}$	4
***C:**	**FU1**		1
D:	FU1		4
E:	FU1		4
F:	FU1		4
G:	FU1		4
H:	FU2		4

(b) Input from Example 3

Figure 4.5: Two anomalous inputs found from the setting



(a) Trace from Example 2 (b) Trace from Example 3

Figure 4.6: Two TA traces found by the framework

The only essential difference between Examples 2 and 3 is the length of the misprediction region. The scheduling of instructions A, B, and H remains identical in both examples. This suggests that certain anomalies may share underlying mechanisms, potentially enabling a classification of timing anomalies (TAs) based on their structural similarities.

Another notable observation is that, in these cases, the anomalous effect can be attributed solely to the delayed fetch of instruction H. Remarkably, an equivalent effect can be reproduced by introducing a cache miss during the fetch of H, as illustrated in Figure 4.7. In this scenario, both α and β traces feature correct branch prediction; however, a cache hit occurs in α, while a cache miss occurs in β. This equivalence may help us to adapt the existing definition.

TODO: **TA only on commit; no longer effect?**

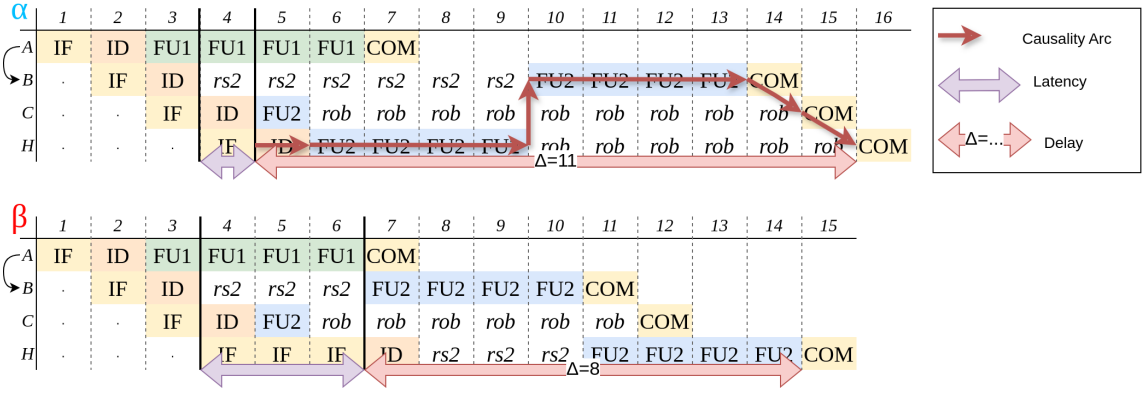


Figure 4.7: Cache miss on fetch of H causing the same effect as branch misprediction

4.4 Formalizing TA Definition

TODO: **introdcue letters for instructions to make things clear**

Given that, for the Examples 2 and 3, there exists an equivalent scenario without branches in which the timing anomaly can be explained by Binder's definition (see Figure 4.7), it appears feasible to apply the same definition to branch-related cases with minimal modification.

To adapt the existing definition in the context of branch predictors, we start by defining a variation, which is the source of a TA and therefore should be understood first. First, let us consider the trace illustrating the equivalent behavior (Figure 4.7) to Example 2, where the variation occurs in the IF latency, i.e., the delay between $\uparrow \mathrm{IF}_H$ and $\downarrow \mathrm{IF}_H$. In Example 2, the event $\downarrow \mathrm{IF}_H$ has the same positions in traces α/β as in the equivalent trace (Figure 4.7), so it can be used as the end of the variation. But for the start of the variation, $\uparrow \mathrm{IF}_H$ cannot be used in the case of branching, as $\uparrow \mathrm{IF}_H$ moves along with $\downarrow \mathrm{IF}_H$, keeping the same delay. Notably, in Binder's framework, for a given instruction, $\uparrow$ IF coincides with the $\downarrow$ IF of the preceding instruction. This means that we can use $\uparrow$ IF and $\downarrow$ IF of consecutive instructions interchangeably during latency definition. Therefore, the latency can be measured between the $\downarrow$ IF of the branch instruction and the $\downarrow$ IF of the first instruction following the misprediction region.

However, it is important to account for other variations that may occur concurrently, as variations in the IF stage of the post-branch instruction can overlap with branch-related variations, potentially confounding their respective latencies. To address this, we propose marking two distinct events to capture the variation:

1. **Branch Prediction (BP)** – the moment when the prediction occurs and speculative execution starts;

2. **Correct Branch Taken (BT)** – the instruction from the correct branch enters the pipeline.

Note that *BP* corresponds to the $\downarrow$ IF of the branch instruction and *BT* is the $\uparrow$ IF of the first instruction after the misprediction region of the branch. In the case of a single variation that is in branch prediction, the TA pattern is detected in the same way as shown in Figure 4.7, but with the latency being shorter by one clock cycle and the delay being longer by the same value.

Figure 4.8 shows this idea applied to Example 2. Both latencies corresponding to the variation are one clock cycle shorter compared to the example shown in Figure 4.7. The causality region also starts one clock cycle earlier; however, it still goes through the same events. Note here that we do not need any additional rules to build causality arcs.

A major assumption that allowed us to adapt the definition is that the branch behavior is just postponing the fetch of the instruction following the misprediction region. In this case, the behavior can be compared to the effect of a cache miss in the IF stage. This is also not completely true, as the penalty for a cache miss is fixed, while the size of the misprediction region depends on the time between the prediction and the resolution of the branch. Therefore, in the rest of the article, we focus more on examples that do not fit this assumption.

TODO: **Bound the effect of branch resolution?**

TODO: **Conclusion: definition somehow works, but we need to study its limitations**

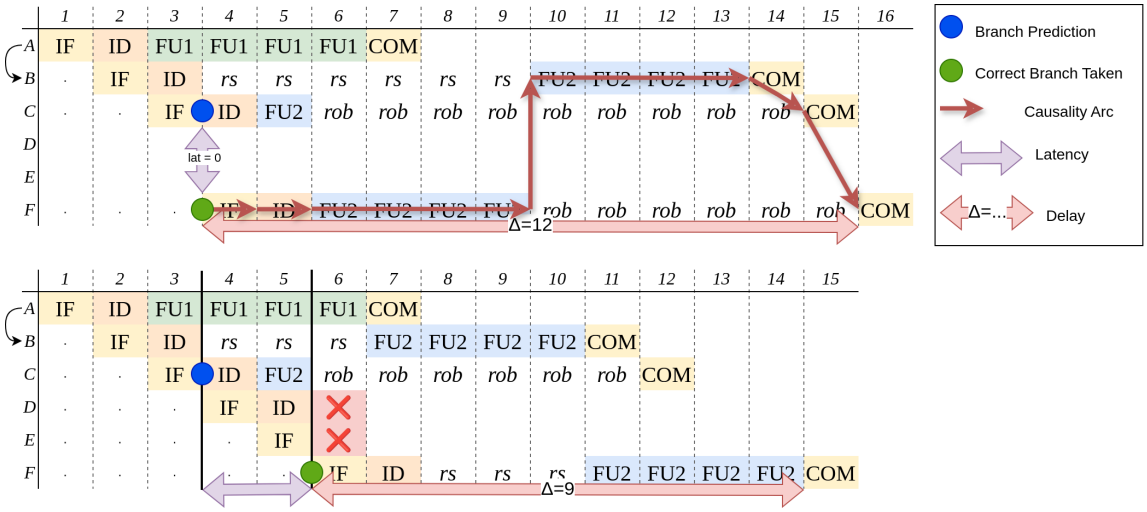


Figure 4.8: The new TA-definition applied to Example 2

4.5 Limitations of the Definition

So far we have demonstrated the potential of using Binder's definition in detecting branch prediction-related TAs. Now we discuss the limitations of the method by providing the relevant examples. We start from describing the problem that we found that appears within the existing framework and is not even related to branch prediction. Then we give more complex examples of branch prediction-caused TAs and try to reason whether they exhibit the same type of flaw or some new issues.

4.5.1 Gap Problem

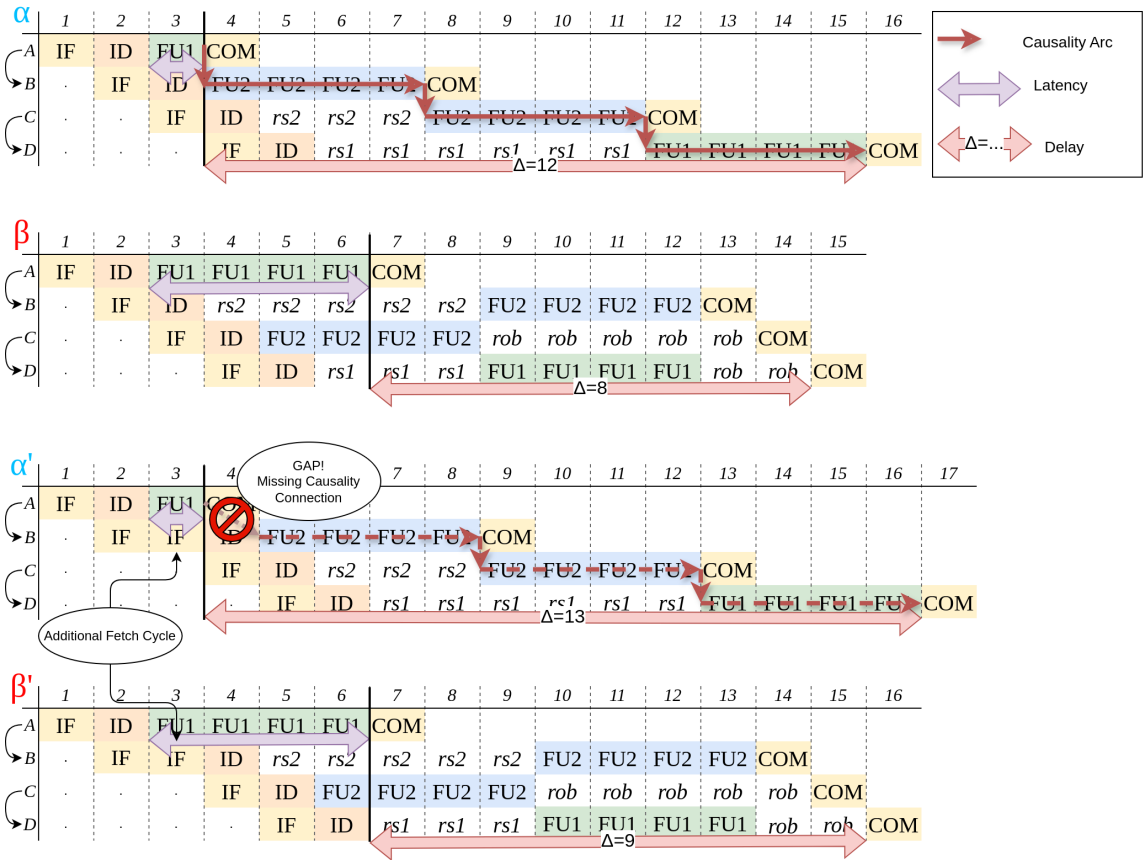


Figure 4.9: Gap problem

While testing an existing definition using framework made by Binder et al. we encountered a situation when TA clearly exists but is not captured by the framework. Such an issue we called the *gap problem*. Figure 4.9 shows an example of such a problem: pair of traces α/β shows and example of TA similar one seen in Example 1, Binder's definition works fine for this pair. In a pair of traces α'/β' we change a latency for fetch of instruction B. This creates a so-called *gap* between $\downarrow$ FU of A and $\uparrow$ FU of B: a situation where a ETDG arc does not become a causality arc. Here it happens because a more strong causality link $(\downarrow \mathrm{IF}_B) \rightarrow (\uparrow \mathrm{FU}_B)$ is established using the rule of the most relevant constraint. Therefore, there is no causality path between end of the variation $\uparrow \mathrm{FU}_A$ and COM_D, so the timing anomaly is not signalled.

However, one can argue, that TA exists in α'/β' as we observe a slowdown in the trace α' with a favorable variation. Moreover, the two pairs of traces share the same suffixes and the scheduling pattern that leads to TA is the same. The only thing that prevents the TA from being detected is the absence of causality path between $\downarrow$ FU of A and $\uparrow$ FU of B in trace α'.

The understanding of the flaws of the definition Binder's definition [2] allows us to reason about further results in terms of gap problem: following we describe some examples of branch prediction TA examples that do not fit into causality-based approach and identify whether it is the same type of problem or not.

4.5.2 Early FU release

Up to this point, we have considered branch predictor-related timing anomalies (TAs) arising from the postponement of the fetch of the first non-mispredicted instruction. This occurs when no instructions within the misprediction region are in the execution phase, and thus the region does not affect functional unit (FU) scheduling. In the following examples, we analyze the impact of the misprediction region on instructions that were fetched prior to the branch.

Example 4:

Nested Misprediction Region

We consider the instruction trace shown in Figure 4.11, which features two nested misprediction regions corresponding to instructions C and D. We analyze the effect of varying the prediction outcome for D: in trace α (Figure 4.11), the prediction is correct, resulting in a longer overall execution time compared to trace β, where the prediction is incorrect.

	Res.	Dep.	Lat.
A:	FU1		5
B:	FU2	$\{A\}$	4
C:	FU1		1
**D:*	FU2		1
E:	FU2		4
F:	FU2		4
G:	FU2		4

Figure 4.10: Instruction trace from Example 4

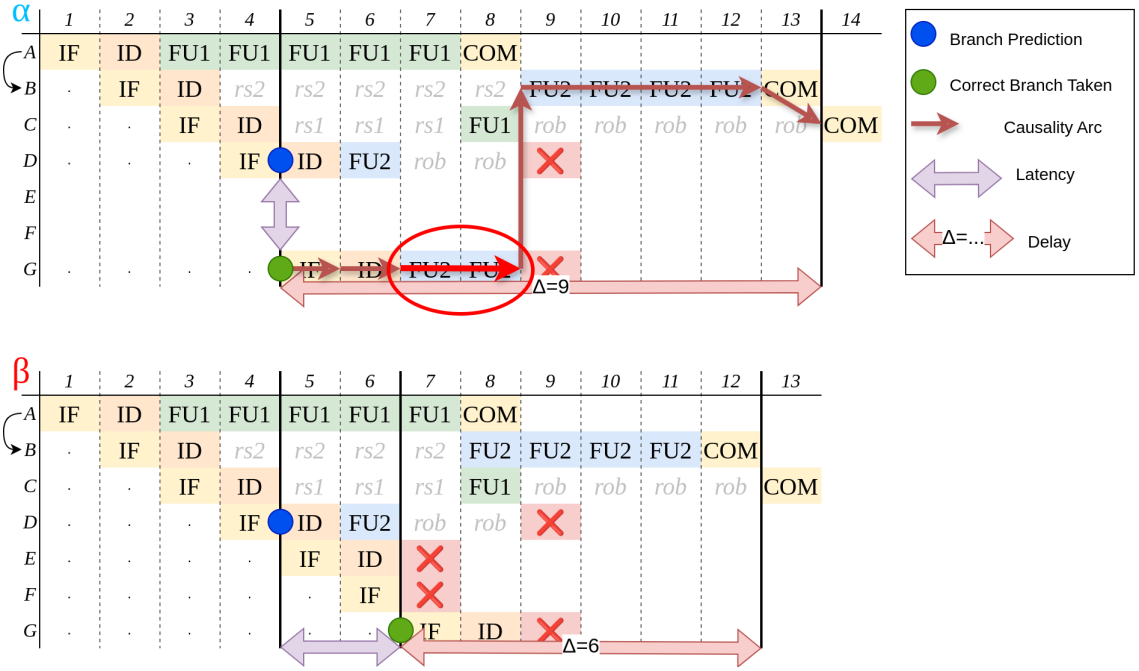


Figure 4.11: Execution trace of Example 4

In Figure 4.11 we try to construct a causality graph for trace α. On the picture we show the causal path form variation to the commit of C. However, one causal arc on the path (encircled in red) does not fit the definition: Remember that in Binder's definition we have the *rule 4* which creates a ETDG-arc ($\uparrow \mathrm{FU} \xrightarrow{lat_{FU}} \downarrow \mathrm{FU}$). In Example 4 the FU-latency of G is 4, however in trace α the delay between $\uparrow \mathrm{FU}_G$ and $\downarrow \mathrm{FU}_G$ is only 2 clock cycles because of the earlier release of FU2 as the result of squashing. So according to the Binder's definition the arc cannot be established and not TA exists.

Likely, we are just missing a rule that suffices to describe anomaly here. Intuitively, $\downarrow \mathrm{FU}_G$ is caused by the squashing triggered by the branch resolution. Therefore, ($\uparrow \mathrm{FU}_C \xrightarrow{0} \downarrow \mathrm{FU}_G$) arc would make sense (Assumption 1). However, it does not allow to construct a causality path in Example 4 that can explain the TA happening.

Assumption 1:

If FU is released as the result of squashing, the FU release of corresponding branch is causal to FU release of the squashed instruction.

Indeed, what we see in this example is that the earlier fetch of G allows it to start executing, which prevents B from starting the execution earlier. In fact, it is a reordering problem similar to Example 2. For that reason we take an Assumption 2 that effectively fills the gap in the causality path.

TODO: **why this is not a gap problem?**

Assumption 2:

The FU acquisition is always causal to the respective FU release.

However, the following example shows that Assumption 2 is not always correct.

Example 5:

Latency Impact Through Misprediction Region

We consider an instruction trace from Figure 4.12 the pair of execution traces with TA is shown at Figure 4.13. Here the variation is in latency of B, but the effect propagates through misprediction region of F which consists of one instruction G.

	Res.	Dep.	Lat.
A:	FU3		9
B:	FU1		6|7
C:	FU2	$\{A, B\}$	4
D:	FU1	$\{C\}$	4
E:	FU2	$\{B\}$	4
F:	FU1		1
G:	FU2		4

Figure 4.12: Instruction trace of Example 5

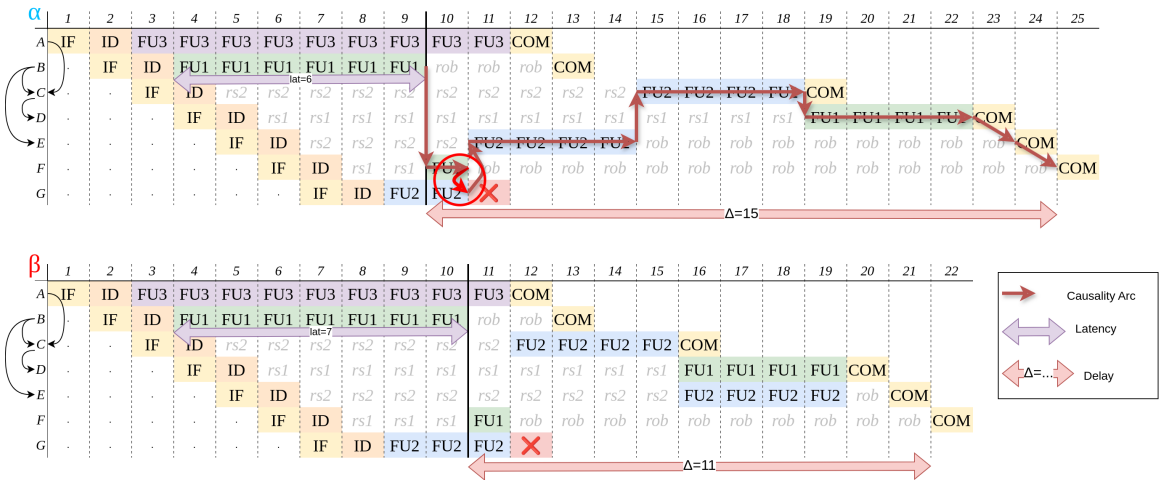


Figure 4.13: Execution trace of Example 5

Example 5 illustrates the problem with Assumption 2: here if we try to draw a causality arc ($\uparrow \mathrm{FU}_G \xrightarrow{2} \downarrow \mathrm{FU}_G$) in trace α, the causality path can not build between $\downarrow \mathrm{FU}_B$ and COM_E. Instead, the Assumption 1 works here allowing us to draw an arc ($\downarrow \mathrm{FU}_F \xrightarrow{0} \downarrow \mathrm{FU}_G$).

TODO: **problem: consider only one trace**

TODO: **Conclusion: we need another definition**

Example 6:

Branch Prediction + Gap problem

We modify Example 4 by increasing latencies of instructions A and C by 2 (input trace in Figure 4.14). This also requires us to define a larger misprediction region. The resulting pair of traces (Figure 4.15) exhibit a similar TA: instruction J is fetched in trace α and postpones B.

	Res.	Dep.	Lat.
A:	FU1		7
B:	FU2	{*A*}	4
C:	FU1		1
**D:*	FU2		3
E:	FU2		4
F:	FU2		4
G:	FU2		4
H:	FU2		4
I:	FU1		4
J:	FU2		4

Figure 4.14: Instruction trace of Example 6

Example 6 shows that the gap problem can also arise in misprediction region. $\downarrow \mathrm{FU}_J$ in trace α is not detected as a part of causal region of the end of the variation $\uparrow \mathrm{IF}_I$ because of the resource contention rule which creates a stronger arc ($\downarrow \mathrm{FU}_D \xrightarrow{0} \uparrow \mathrm{FU}_J$).

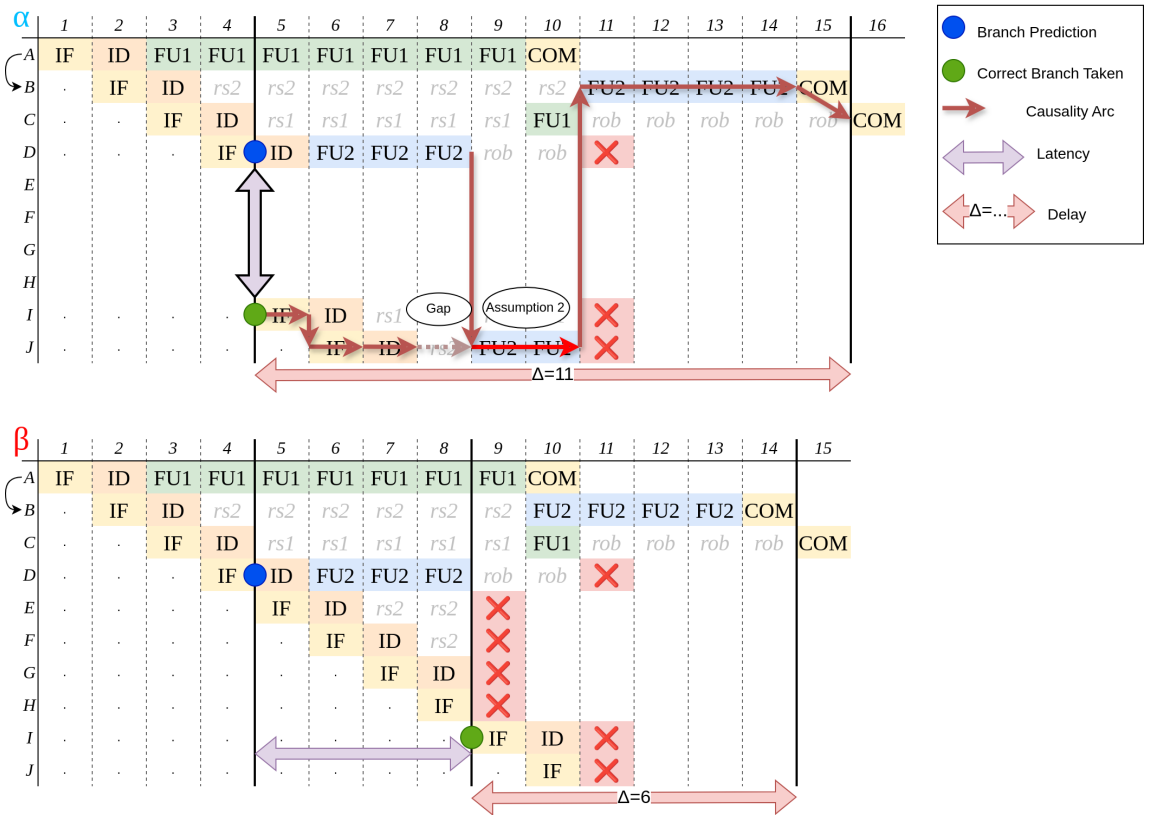


Figure 4.15: Execution trace of Example 6

In summary, our analysis demonstrates that while the definition of Binder et al. shows promise for application to branch prediction scenarios, its adaptation requires the introduction of additional assumptions. However, the two assumptions we considered are mutually contradictory, precluding the formulation of a consistent and comprehensive definition. Furthermore, the original definition is susceptible to the gap problem, which also manifests in the context of branch prediction-induced timing anomalies.

We argue that these issues arise from fundamental limitations in the notion of causality adopted in Binder's work. Intuitively, causality should capture relationships such as "event X was delayed due to event Y," but the current definition constructs causality arcs based solely on a single trace, without considering differences between traces. In Binder's framework, timing dependencies present in the trace with the unfavorable variation are disregarded, which is critical for understanding the propagation of timing anomalies. Both the gap problem and the contradictions between the proposed assumptions appear to originate from these conceptual shortcomings in the causality definition. Consequently, we believe that substantial revisions to the underlying notion of causality are necessary.

— 5 —
Conclusion

In this study, we developed a framework to investigate branch prediction-induced timing anomalies (TAs). This framework enables the construction of representative examples, allowing us to systematically explore the phenomenon. Utilizing this tool, we demonstrated that the definition proposed by Binder et al., which we identified as the most prominent among existing definitions, can be adapted to scenarios involving branch prediction. Nevertheless, our analysis revealed controversial cases that challenge whether Binder's notion of causality truly captures the intuitive understanding of causality in this context. Consequently, we conclude that further investigation into this aspect is necessary.

There are several features that we did not implement in the current framework, such as a shared memory bus model and limitations on the reservation station (RS) and reorder buffer (ROB). These extensions could be incorporated with minimal changes to the framework. Additionally, we did not address the effects of multiple TAs or how effectively their impacts can be separated using the current definition. However, given the lack of a consistent definition, it is premature to pursue this direction.

Another important aspect not addressed by our model is the state of the branch predictor. In our approach, prediction and misprediction are treated as black-box events, similar to how we abstract cache-induced latencies as non-deterministic. While this abstraction simplifies the model, it overlooks the potential to explicitly represent the branch predictor state. As discussed in Chapter 2, branch predictor requires additional hardware features, such as Pattern History Tables (PHT) and Branch Target Buffers (BTB). They could be modeled as part of the pipeline state, allowing us to consider related events in greater detail. This extension could provide deeper insights into how branch decisions propagate through the system and enable us to study not only individual prediction or misprediction events, but also sequences of such events, potentially revealing new types of timing anomalies.

Bibliography

[1] Ahmet Akkas, Michael J. Schulte, and James E. Stine. *Evaluating the Impact of Accurate Branch Prediction on Interval Software*, pages 69–79. Springer US, Boston, MA, 2001.

[2] Benjamin Binder. Definitions and detection procedures of timing anomalies for the formal verification of predictability in real-time systems.

[3] Florian Brandner. Deliverable D2.1: Study of Timing Anomalies Documented in the Literature, June 2025.

[4] Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. What is a timing anomaly? 23:1–12. Artwork Size: 12 pages, 506419 bytes ISBN: 9783939897415 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] C. Ferdinand. Worst case execution time prediction by static program analysis. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 125–127. IEEE.

[6] Gernot Gebhard. Timing anomalies reloaded. 15:1–10. Artwork Size: 10 pages, 305021 bytes ISBN: 9783939897217 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[7] Alban Gruin, Thomas Carle, Christine Rochange, Hugues Casse, and Pascal Sainrat. MINOTAuR: A timing predictable RISC-v core featuring speculative execution. 72(1):183–195.

[8] Sebastian Hahn and Jan Reineke. Design and analysis of SIC: A provably timing-predictable pipelined processor core.

[9] W.H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, 1977.

[10] Christopher Healy, Mikael Sjödin, and Viresh Rustagi. Bounding loop iterations for timing analysis. pages 12–21, 01 1998.

[11] Raimund Kirner, Albrecht Kadlec, and Peter Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 119–128. IEEE.

[12] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley.

[13] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, pages 12–21. IEEE Comput. Soc.

[14] Nick Mahling. Reverse engineering of intel's branch prediction.

[15] Arthur Perais. Increasing the performance of superscalar processors through value prediction.

[16] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times — a graph-basedapproach. *Real-Time Syst.*, 13(1):67–91, July 1997.

[17] Bjarne Stroustrup. *The C++ programming language: C++ 11*. Addison-Wesley, 4. ed., 4. print edition.