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

Exploration by model-checking of timing anomaly cancellation in a processor

Andrei Ilin

Defense Date, 2025

Research project performed at VERIMAG

Under the supervision of:
Lionel Rieg

Florian Brandner

Mihail Asavoae

Defended before a jury composed of:
Head of the jury
Jury member 1
Jury member 2

June 2025

Abstract

Your abstract goes here...

Acknowledgement

I would like to express my sincere gratitude to .. for his invaluable assistance and comments in reviewing this report... Good luck :)

Résumé

Your abstract in French goes here...

Contents

— 1 —
Introduction

This chapter is not finished yet

1.1 WCET Analysis

In critical systems such as planes and cars it is important that the tasks executed on the hardware meet their deadlines. This is ensured by worst execution time (WCET) analysis. It takes the pair of the program and the dedicated hardware and aims at giving an upper-bound on execution time.

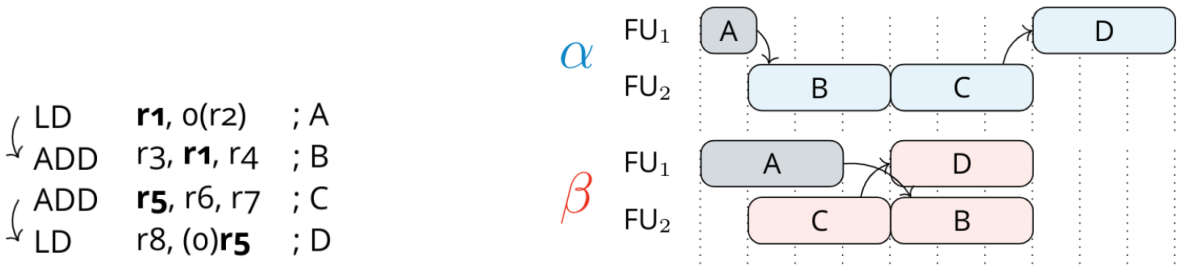
TODO: **stages of WCET-analysis**

1.2 Timing Anomalies

Phase ordering is a major chellange in WCET-analysis. Most of analysis steps require information from each other (TODO: **examples**), so it is not always possible to order them.

Nevertheless, most architectures are not composable and contain so-called timing anomalies (TA). Intuitively, TA happens when local worst cases do not constitute a global worst case. TA is observed on the pair of execution traces where the initial hardware state differs, and the instruction sequences are identical. Different cache states can be the source of variation in timing behavior due to miss in one trace and hit in another one.

Example 1 *Figure 4.3 shows the example of such an anomaly. Here, the assembly sequence consists of 4 instructions* (A,B,C,D) *with data dependencies* $A \rightarrow B$ *and* $C \rightarrow D$*. Figure 1.1b represents the pair of traces* (α,β) *derived from execution of the given program. There is a variation in latency of instruction A (*1 *in* α *and* 2 *in* β*). In trace* α *the variation is favorable, but the total execution time is also higher in this trace which signals an anomaly.*



(a) Inpus assembly sequence

(b) Scheduling on functional units comparison

Figure 1.1: TA caused by variation in latency of instruction *A* (from [1])

— 2 —
Processor architecture

2.1 Instruction Set architecture

Instruction Set Architecture (ISA) defines the set of instructions and the registers on which they operate. From ISA-perspective all instructions are executed atomically, and their result is visible only after an execution is complete. ISA serves as an interface between software and actual hardware microarchitecture which implements the ISA. This abstraction allows software to reason about program behavior independently of the underlying microarchitectural implementation, which may use additional internal registers or buffers and update state at finer (cycle-level) granularity.

ISA defines binary format of instructions which are stored in memory and accessed by the processor through cache mechanisms. Often, fixed length instructions are used (for example, RISC architectures TODO: **reference**), while variable-length also exist (in CISC architectures TODO: **reference**). Processor is a cycled device that performs fetching instructions from memory and their subsequent execution, we call the microarchitectural state the state of all hardware registers of the processor. Unlike in ISA, states are defined at clock-cycle granularity, so an instruction may take several clock-cycles to finish.

TODO: **rewrite**

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 lower 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 multiscalar pipelines, out-of-order execution and branch prediction.

1. **Instruction Fetch (IF)**: The processor retrieves the next instruction from memory by the

address from the program counter (PC). 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.

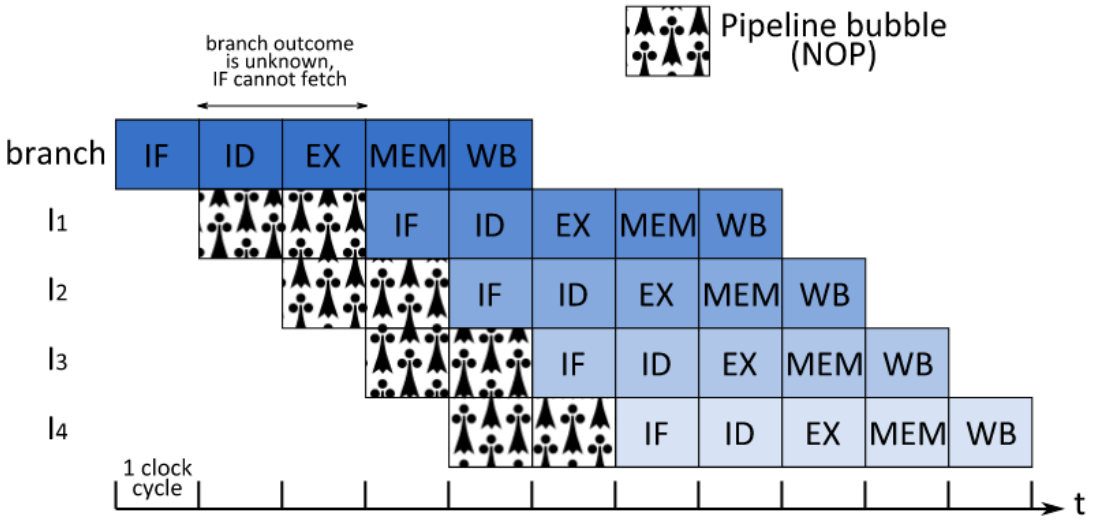


Figure 2.1: Example of control hazard: the pipeline is stalled until branch finished the execution (from [10])

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).

Without bypass network:

	1	2	3	4	5	6	7	8	9	10	11
A	IF	ID	EX	MEM	WB						
B	·	IF	ID	*id*	*id*	EX	MEM	WB			
C	·	·	IF	*if*	*if*	ID	*id*	*id*	EX	MEM	WB

With bypass network:

	1	2	3	4	5	6	7
A	IF	ID	EX	MEM	WB		
B	·	IF	ID	EX	MEM	WB	
C	·	·	IF	ID	EX	MEM	WB

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 Multiscalar Execution

Instead of fetching instructions one by one, it is possible to fetch several ones in the same time. This also means that other stages are also duplicated to accommodate all fetched instructions. Since neighbor instructions may be independent this can significantly increase the performance. However, duplicating each stage is costly, while it is relatively easy for IF, ID and COM, execution and accessing memory is much harder to duplicate.

TODO: **picture**

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 and memory 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 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.

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 2.3 shows the diagram of stage interconnections in processor. It represents the two optimizations discussed: multiscalar and OoO execution.

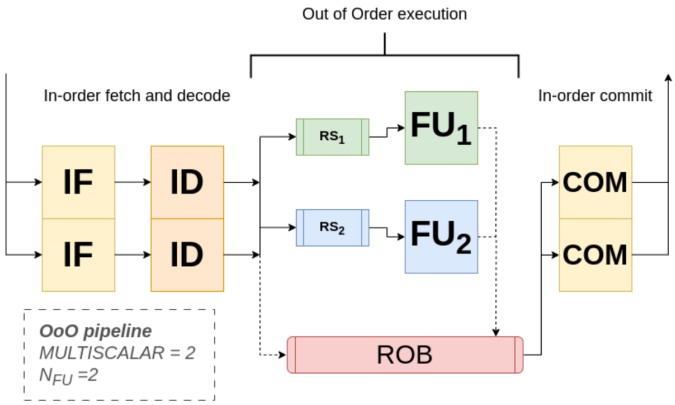


Figure 2.3: Out-of-Order multiscalar pipeline diagram

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 by far is 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.

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.

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 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 2.4 shows the transitions between the states.

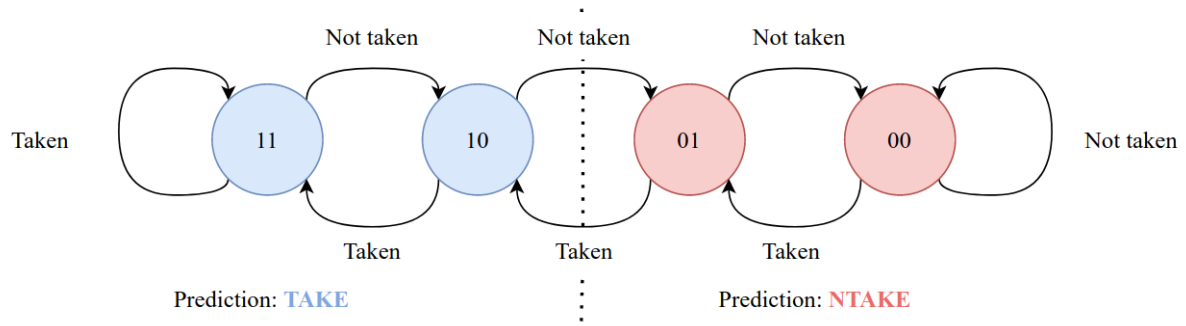


Figure 2.4: Two-bit predictor state machine (from [9])

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.

— 3 —
Timing Analysis and Anomalies

3.1 Evolution of TA-definitions

A timing anomaly (TA) is a situation where a local favorable condition leads to a globally worse state (for example, a cache hit leading to slowdown of the program). The notion of timing anomalies dates back to 1999 when they were first introduced by Lundqvist and Stenström [8] in context of timing analysis. Basically, TA is a feature of an architecture which makes it hard to analyze timing behavior properly. Such anomalies may have a tremendous impact on execution time which is not captured by the WET analysis. Especially dangerous is so-called domino effect, also discovered by Lundqvist and Stenström. It leads to an unbounded slowdown effect of the TA.

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 [4]) and some being more abstract and general (like [1], [5]). In this section we are giving an overview of existing definitions comparing their strengths and weaknesses.

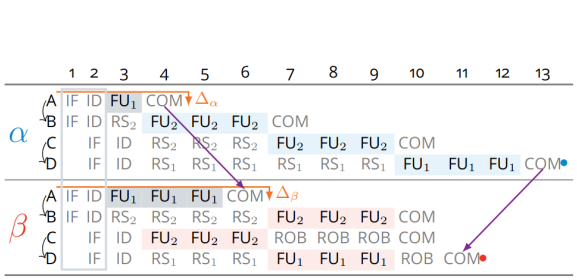
3.1.1 Step Heights

Gebhard [3] gives a timing-anomaly definition based on local execution time of instructions in comparison to global execution time defined as sum of local ones. A TA exists when local execution time of earlier instruction is lower and the global execution time of some later instruction is higher (compared to other trace).

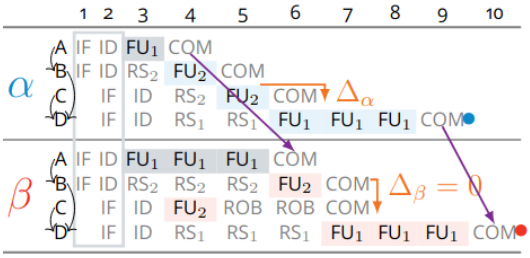
Figure 3.1a shows this definition applied to example 1. Orange arrow illustrates the local execution time of instruction A. The global time for instruction D is different between traces α and β (13 and 11 respectively).

In his thesis [1], 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 [1])

3.1.2 Step-functions Intersections

Similar definition is proposed by Cassez et al. [2]. The difference is that only global execution time is taken into account. Thus, TA arises when step-functions (that map instructions to their absolute completion time) of two traces intersect.

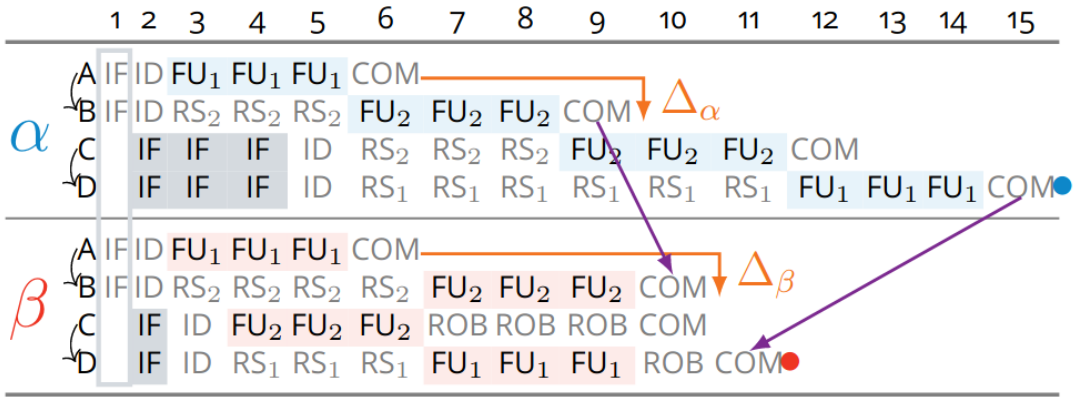


Figure 3.2: Contradicting result of Cassez's definition (from [1])

This definition also leads to misleading effect with scenario found by Binder [1]. Figure 3.2 illustrates this by comparing two traces. 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 IF stage of instructions C and D.

3.1.3 Component Occupation

An alternative approach is proposed by Kirner et al. [6]. In their work the idea is to partition hardware into components and for each define the occupation by instruction (for how many cycles it processes the instruction). TA arises when a shorter component occupation coincides with a longer execution time in a chosen trace. However, as is shown is [1] the results depend on how we define component partition which imposes the major concern against using this definition.

TODO: **counterexample**

3.1.4 Instruction Locality

3.1.5 Progress-based definition

Hahn and Reineke [5] introduce the notion of progress, ... [4]

3.1.6 Event Time Dependency Graph

Binder et al. [1] define TAs using the notion of causality between events in execution trace. In this work, multiscalar OoO pipeline is considered. The processor state is described as a composition of states of each of the resource: *IF, ID, set of RS, set of FU, ROB, COM*. Each component holds the information about instruction it is currently processing, including required registers and remaining clock cycles.

Notion of event is introduced based on qualitative changes in the pipeline associated to instruction progressing through stages. Event from execution trace (denoted as $e \in Events(\alpha)$) is a triple (i,r,t), where i is the instruction to which 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 event occurs.

In the proposed framework events are related to *IF, ID, FU* and *COM* stages. For each instruction there are 7 types of events: $\uparrow$ IF, $\downarrow$ IF, $\uparrow$ ID, $\downarrow$ ID, $\uparrow$ FU, $\downarrow$ FU and *COM*. $\uparrow$ signs the acquisition of a resource and $\downarrow$ its release. *COM* denotes the acquisition of the commit stage; hence its release always happens one clock cycle after and no subsequent stages exist, it is not included into 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 corresponding events. Therefore, 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 $CospEvent : Events(\alpha) \to Events(\beta)$.

A **variation** signs that the latency in one trace differs from latency of corresponding events in the other trace. On the 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 chosen as a source of timing anomalies. They may represent different memory behavior (cache hit or miss) for fetch and memory accesses in FU. Other sources of TA such as memory bus contention or branching are not considered by the framework.

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

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 – 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**

   $(I, \uparrow \text{IF}, t_0) \xrightarrow{lat_{IF}} (I, \downarrow \text{IF}, t_1) \xrightarrow{0} (I, \uparrow \text{ID}, t_2) \xrightarrow{1} (I, \downarrow \text{ID}, t_3) \xrightarrow{0} (I, \uparrow \text{FU}, t_4) \xrightarrow{lat_{FU}} (I, \downarrow \text{FU}, t_5) \xrightarrow{0} (I, COM, t_6)$

   lat_{IF} and lat_{FU} are the latencies of IF and FU stages respectively.

2. **Resource use**

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

3. **Instruction order**

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

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

4. **Data dependencies**

   RAW dependency between I_1 and I_2 (TODO: **dep notation**) 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.

Causality graph is achieved from 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.

Timing anomaly is observed on pair of traces α and β if there exists a favorable variation in α relative to β. Let $e_\alpha\downarrow$ and $e_\beta\downarrow$ 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 causality graph of α between $e_\alpha\downarrow$ and e_α, s.t. $\Delta(e_\beta\downarrow, e_\beta) < \Delta(e_\alpha\downarrow, e_\alpha)$.

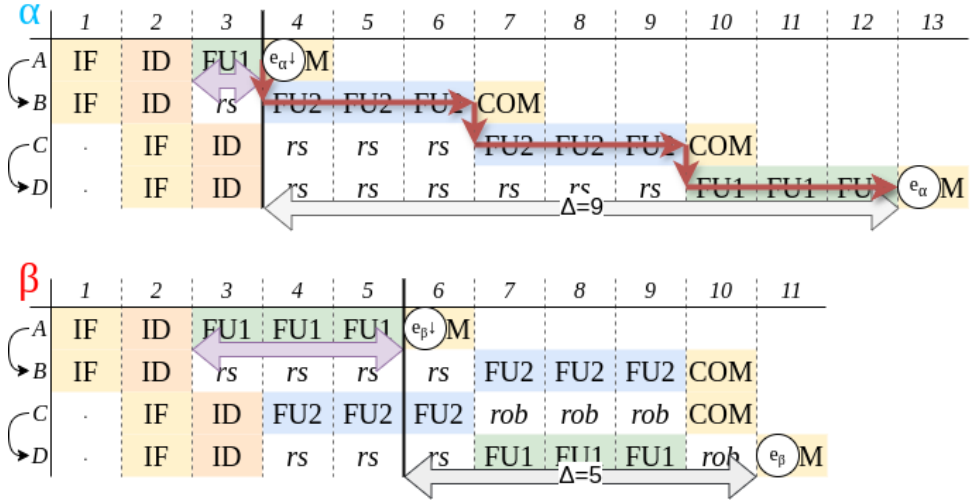


Figure 3.3: Causality-based TA detection applied to Example 1. $e_\alpha\downarrow = (A, \downarrow \mathrm{FU}, 4), e_\beta\downarrow = (A, \downarrow \mathrm{FU}, 6), e_\alpha = (A, COM, 13), e_\beta = (A, COM, 11)$. Purple arrow denotes latency which has a variation between two traces. Gray arrow shows delay between events which is greater in favorable trace. Causality in path α is marked by red arrows.

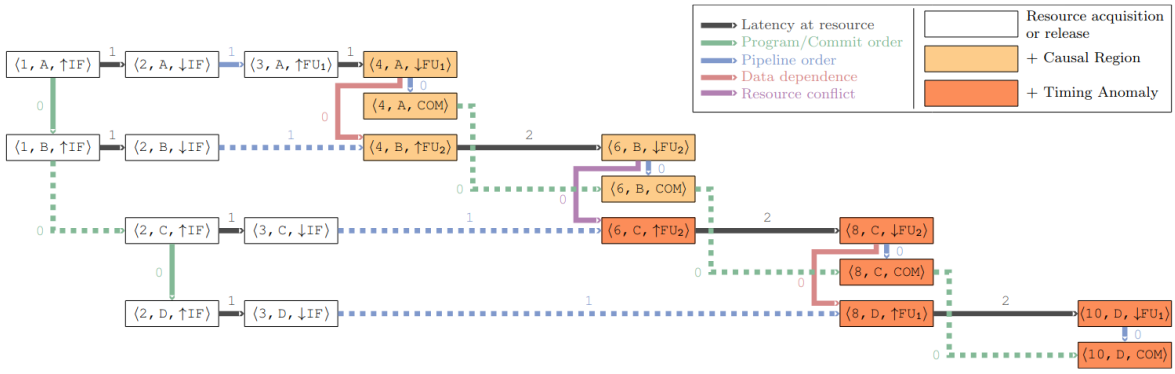


Figure 3.4: Complete ETDG for trace α from Figure 3.3, trace α. TODO: **image source**

Figure 3.3 shows how the framework captures TA for example 1. Figure 3.4 presents the complete ETDG for trace α with different dependency rules highlighted with different colors. The arcs reflecting causality are depicted in solid lines.

In contrast to other definition, 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 TA-effect.

3.2 TA-classifications

— 4 —
Contribution

Definition of Binder et al. [1] is promising, however the branch prediction and the related issues are not taken in account by the framework. Thus we aim to extend the use case of proposed definition.

In our work we try to adjust Binder's definition to the setting of pipeline with branch predictor. We introduce an input format capable of expressing speculative execution.

TODO: **complete intro when chapter is done**

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$^+$ [7] 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$^+$ [7]. The pipeline state is specified in set-theory notation. The model checker step corresponds to a one clock cycle and derives a new HW state from the previous one. This allows to simulate the non-deterministic timing behavior: each time when a variation can happen, multiple next state are generated. TLA$^+$ covers all reachable states ensuring that all possible behaviors are covered.

The pair of trace constitutes a whole model state. TA is expressed as an invariant for the pair of traces, so its is verified in each model checking step.

As well as a 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: superscalar degree, FU latencies and memory access latencies depending on the cache events (hit or miss). sequence of instructions;

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

Figure 4.1 illustrates the instruction sequence that causes TA in Example 1. We can simplify this view by directly expressing the resource, dependencies and possible latencies of instruction. Figure 4.2 shows the input for instruction trace from example 1. First column is instruction label, second is the resource used, thirds 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 IF stage, but we skip them for simplicity. This table is sufficient to express a pair of execution traces derived from 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 1.1a

4.2.2 Limitations

Despite using a model checker, the existing framework is capable to explore only the traces that fit the instruction template. This limits the explored space to what is manually defined by the user. Considering that branches are to be added, this limitation is becoming even more restricting.

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

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

Moreover, the input is specified in lengthy TLA+ notation, which prevents fast sketching the 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 instruction trace format is straightforward: it specifies all instructions that are fetched, executed and finally committed. In 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 *misprediction region of branch instruction*.

As an input trace we specify all instructions that can enter the processor pipeline. As we focus only on timing behavior of the program, abstracting from memory and registers 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 case of correct prediction. However, in 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

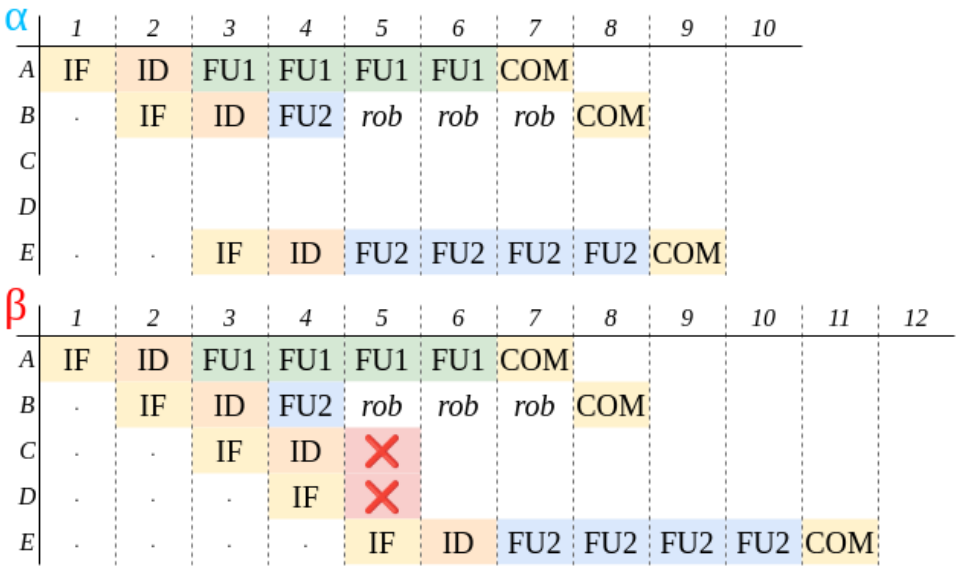


Figure 4.4: Pair of traces with correct and incorrect predictions. The squashing event is denoted with a red cross.

TODO: **nested mispred region**

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

Framework Implementation

We decided to take C++ [11] as an implementation language as it is fast and includes a number of useful data structures in a standard library.

We define a single instruction as follows. It consists of type of FU to be scheduled at (we do not consider resource switch), latency in this FU, set of RAW dependencies. If instruction is a branch, `mispred_region` is set to a positive value n denoting the next n instructions are in misprediction region of current instruction. If we want to model only the correct prediction, then `mispred_region` is set to 0; this way branch behaves as an ordinary instruction. `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;

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

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

TODO: describe the setting, the search space given to the tool to produce an example (comment on performance "example was found in ... ms", "... examples were found"). Explain why examples show TA

We start with generating representative examples of branch-caused TA. The hypothesis is that the correct branch prediction may yield a longer execution than the same setting with misprediction. We want to have a simple example, thus decided we perform an exhaustive search in a state of the programs with:

1. 4 committed instructions with;
2. At most 2 dependencies;
3. 1 branch instruction.

The latency of branch instruction was chosen to be 1 as conditionals jumps often require simple one-cycle operations (such as equality, more or equal, less or equal, etc.). The latencies of other instructions in the chosen setting are 4. We have chosen single-scalar pipeline with 2 FUs.

Our framework explored 4608 input traces in around 150 ms: among those 2 TAs were identified, explained in Examples 2 and 3.

Example 2 *Figure 4.5a shows an instruction trace found from generating random examples. C is a branch with misprediction region D, E. There is one dependency: $A \rightarrow B$. Figure 4.6a shows the associated pair of execution traces. In trace α instructions D and E as skipped due to the correct prediction, so F is fetched right away. This causes an earlier execution of F, which leads to $FU2$ being busy at clock cycle 7 and therefore instruction B starts being executed later, thus causing a slowdown.*

Example 3 *The other TA which input and execution trace are shown in Figures 4.5b and 4.6b respectively is different from Example 2 only by the FU of instruction C while the other instructions are the same (also the misprediction region is longer due to longer delay between branch prediction and branch resolution).*

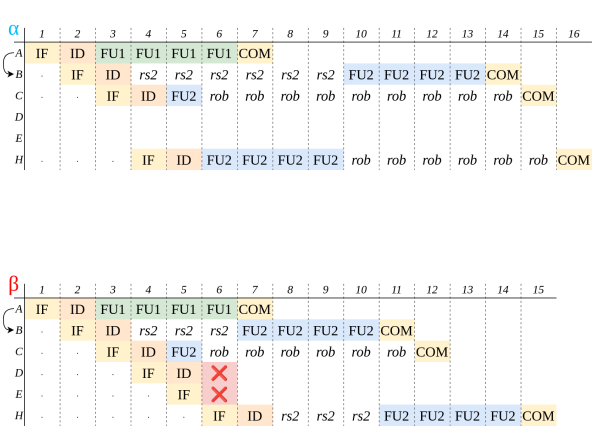
	Res.	Dep.	Lat.
A:	FU1		4
B:	FU2	$\{A\}$	4
**C:*	**FU2**		1
D:	FU1		4
E:	FU1		4
H:	FU2		4

(a) Input from Example 2

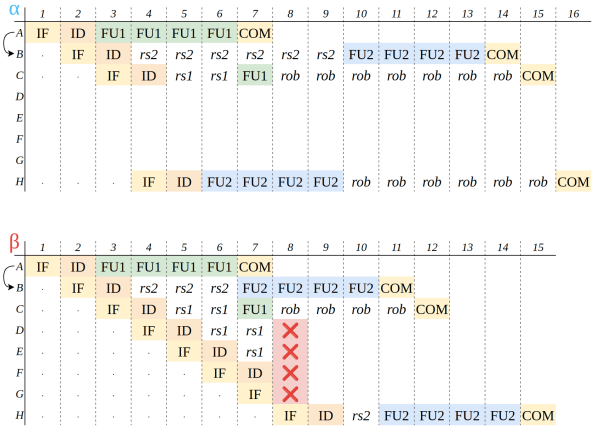
	Res.	Dep.	Lat.
A:	FU1		4
B:	FU2	$\{A\}$	4
**C:*	**FU1**		1
D:	FU1		4
E:	FU1		4
F:	FU1		4
G:	FU1		4
H:	FU2		4

(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 misprediction region. The scheduling of instructions A, B and H is exactly the same in the two examples. This gives us a hint that some anomalies may fall down in the same category which can give us a classification of TAs.

Another notable observation is that here the anomalous effect can be explained by just by the later fetch of instruction H. Interestingly, the same effect can be obtained by modeling a cache miss for the fetch of H as shown in Figure 4.7. Here, int both α and β the prediction is correct, however, there is a cache hit in α and cache miss in β.

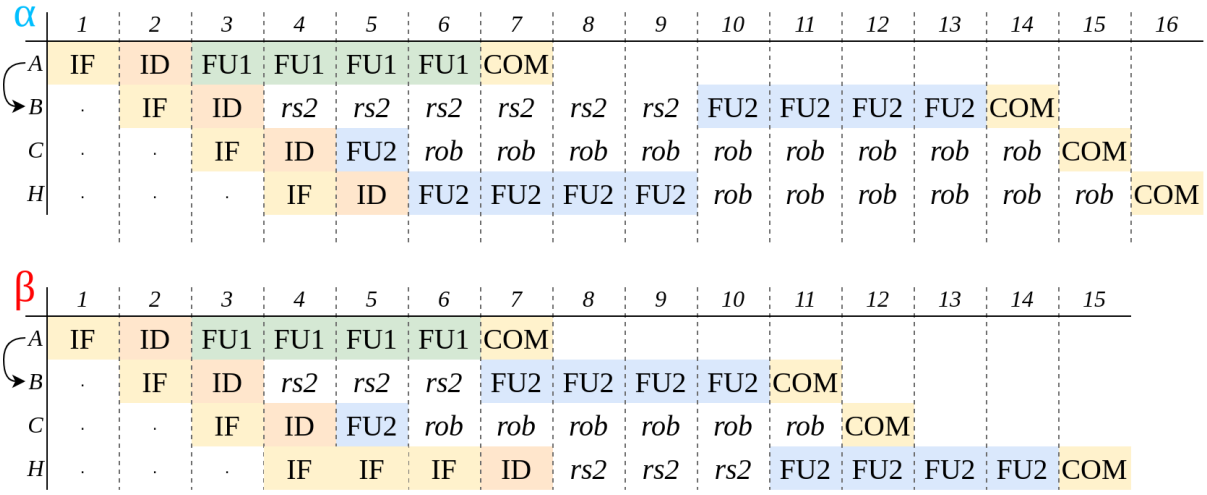


Figure 4.7: Cache miss on fetch of H causing the same effect as branch misprediction

4.4 Formalizing Definition

TODO: start with latency, show it on example, formalize an assumption: show new contradicting examples

For ex 2 3 BInder's def works ...

4.5 Gap problem

4.5.1 Problem statement

TODO: explain that even without branching there is a problem

4.5.2 Requirements for Causality Graph

TODO: formalize properties that we expect from causality graph and explain why they are violated

4.5.3 Towards New Causality Definition

TODO: propose a method that can be used to construct a "true causality" graph: set of constraints, moving events around

— 5 —
Conclusion

Bibliography

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

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

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

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

Appendix

Appendix goes here...