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

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

For real time systems it is important to satisfy timing requirement, meaning that the time of program execution must be predictable. WCET analysis aims at giving an upper-bound of execution time. ... Some text [9]

— 2 —
Background

2.1 Instruction Set architecture

Instruction Set Architecture (ISA) defines the set of instructions and the registers on which they operate. Normally, the instruction operands are read from the registers and the execution result is stored there. ISA serves as an interface between software and actual hardware microarchitecture which implements the ISA.

TODO: **what is ISA-state, instructions-granularity. ISA-level reigsters, mapping to real regs**

2.2 Microarchitecture

ISA defines binary format of instructions which are stored in memory and accessed by the processor through cache mechanisms, usually, fixed length instructions are used, while Ð¼ariable-length also exist. (TODO: **add examples**). 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 takes several clock-cycles to finish. Different optimizations, such as pipelining, multiscalar execution, out-of-order execution and branch predictors (speculative execution).

2.2.1 Processor Pipeline Stages

Each instruction needs several microopererations to be executed: first, the instruction is to be loaded from memory, the operands need to be loaded from the registry. After that the instruction is executed during several cycles depending on its type (for example, multiplication is longer than addition). Due to the fact of isolation of those micropererations, it is possible to execute several instructions simultaneously: when instructions free its stage, next instruction enters it. This optimization, called pipelining, allows to increase the throughput of the processor.

TODO: **Is the word "microopererations" correct here?**

Several decompositions can exist for modern processors. Here we describe the 5 stages that can found in any processor and some of which may be further decomposed in more sophisticated architectures.

Instruction Fetch (IF)

As it was said before, the program instructions reside in global memory. This means that instructions access needs to be performed through memory hierarchy using program counter (PC) address. Often, a special instruction cache exists for accessing the program. IF stage is also responsible for updating PC to read the new instruction.

Instruction Decode (ID)

Once the instruction is fetched from memory, it exists in a processor in a packed binary format. This encoding includes the type of instruction as well as the registers it operates with. Decode stage loads the actual values from Physical Registry File (PRF) and propagates them to downstream pipeline stages. Sometimes the value can be obtained through bypass network before it appears in PRF.

Execute (EX)

EX stage computes the result of the operation. Several components may be responsible for performing different types of operations (for instance, different components for addition and multiplication). In this case IF stage emits control signals that determine the data path.

In case of memory or jump instruction the address is calculated.

The result of the computation is directly available to the ID stage via bypass network.

Access Memory (MEM)

This stage performs access to the global memory through memory hierarchy. If instruction is not a memory instruction, this stage is skipped.

Commit (COM)

The purpose of the last stage is to write the result of the instruction to PRF. Only after this the result is visible from ISA-state perspective.

2.2.2 Restrictions

The structure of the program imposes limitations on execution. Instruction may block each other thus stalling the pipeline.

Data Hazards

There exist three types of register dependencies that may cause pipeline stall.

Read-After-Write (RAW) dependencies, also called as true data dependencies, arise when to perform one operation, the result of the other must be obtained. For example expression $(1+2*3)$ requires $(2*3)$ be calculated first, thus creating *RAW*-dependency between multiplication and addition operations.

Write-After-Write (WAW) dependency happens when two instructions are writing to the same ISA-level register. The two writes must happen is instruction order.

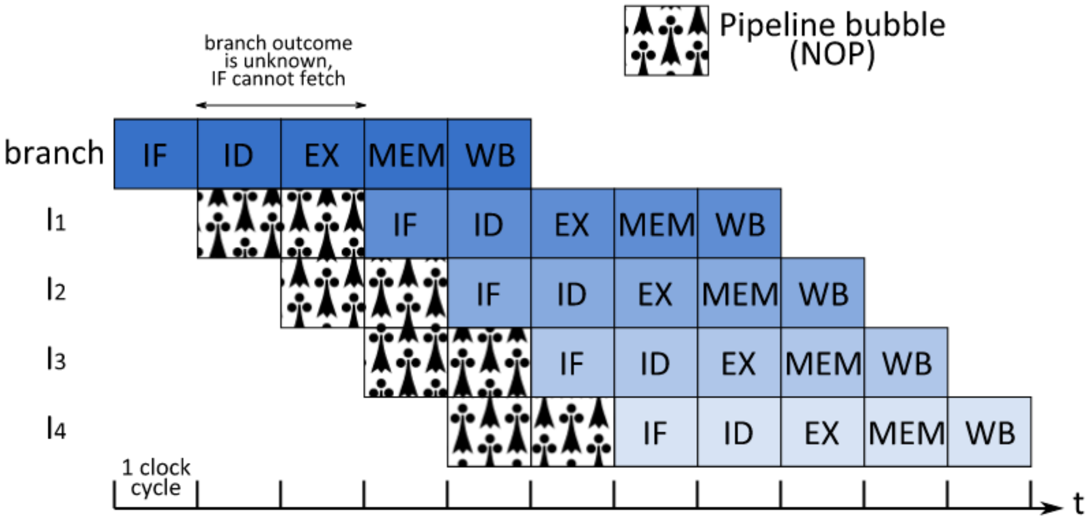


Figure 2.1: Example of control hazard: the pipeline is stalled until branch finished the execution (from [9])

Write-after-Read (WAR) dependency exists when the younger instruction aims at writing a value in the register which is to be read by an older instruction.

RAW-hazards are inevitable in any architecture. WAW and WAR dependencies do not exist in the model we described so far, by must be resolved in out-of-order pipeline.

Control Hazards

Fetching next instruction is possible only if the address of it is known. In case of branches, the next instruction address is not known until the branch outcome is calculated in execute stage.

Therefore, the so-called bubbles (which denote the absence of operation) are introduced into the pipeline.

2.2.3 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 multiplied 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.

2.2.4 Out-of-Order (OoO) Pipeline

Despite the fact that the instructions are to be processed in program order, many of them are in fact independent. This means that the order of execution can be chosen based on instruction dependencies rather than their order in initial program. Notice that the pipeline is often stalled by the execution of long instructions (TODO: **refer visual example**). The key idea is that while one instruction is being executed on one functional unit (FU), the other, independent of this one can be executed on the other FU.

In this approach we divide the pipeline into in-order and out-of-order parts. In-order consists of IF, ID and COM stages while out-of-order includes execution and memory accesses. This allows to achieve a consistent ISA-stage due to in-order fetch a commit.

Different mechanisms exist to synchronize out-of-order execution. Here we introduce reservation stations (RS) and reorder buffer (ROB) - the additional pipeline stages.

Reservation station is a queue before the functional unit, each FU is equipped with its own RS. Once the instruction is decoded it is forwarded to FU based on its type, but if FU is busy, the instruction is put instead into the corresponding RS. Subsequently, the FU is taking the instructions both from ID and RS based on the scheduling policy.

ROB is a FIFO queue that insures the order in which instructions should be committed. Each time, the instruction enters out-of-order part (RS or FU) it is also appended to the front of ROB. After being executed, the instruction is tagged as ready in the ROB. The COM stage commits only the last instruction (or several if multiscalar) from the ROB if it is ready, thus ensuring commit in program order.

TODO: **Image**

2.2.5 Branch Prediction

IF stage is responsible for fetching the next instruction in the program. However, when conditional jump instruction is fetched the next read address is undefined until the outcome of condition is calculated. The straightforward approach is to stall the pipeline, introducing so-called bubbles (no operation).

The more advanced approach consists of fetching a new instruction anyway, the address of which is guessed by branch prediction mechanism, discussed further. Such instructions are called speculative and are not committed until branch decision is taken. In case of incorrect prediction speculative instruction are flushed from the pipeline.

2.3 Branch Predictor Implementations

2.3.1 Static Branch Predictors

Static branch prediction relies on information known at compile time. Some well-known static branch predictors are:

- Always Not Taken
- Always Taken
- Backward Taken, Forward Not Taken

TODO: **add details**

2.3.2 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.

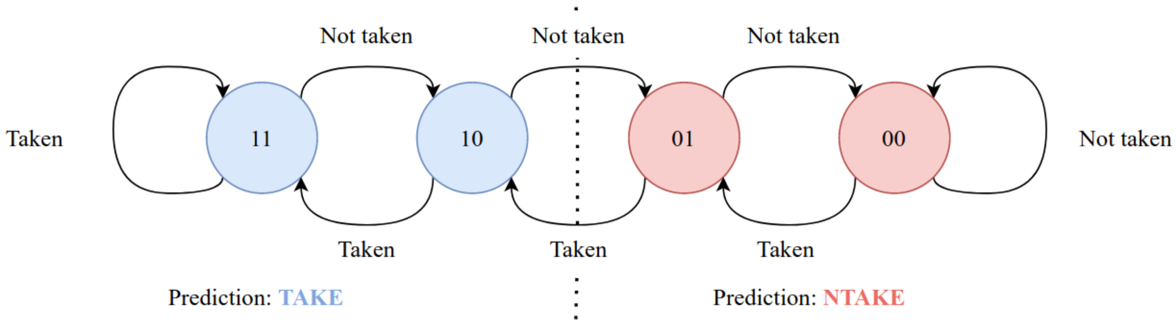


Figure 2.2: Two-bit predictor state machine (from [8])

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. Picture TODO: shows the transitions between the states.

TODO: **why better than 1-bit**

TODO: **other types. which are used in critical systems?**

2.4 WCET Analysis

In critical systems such as TODO: **examples** 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.

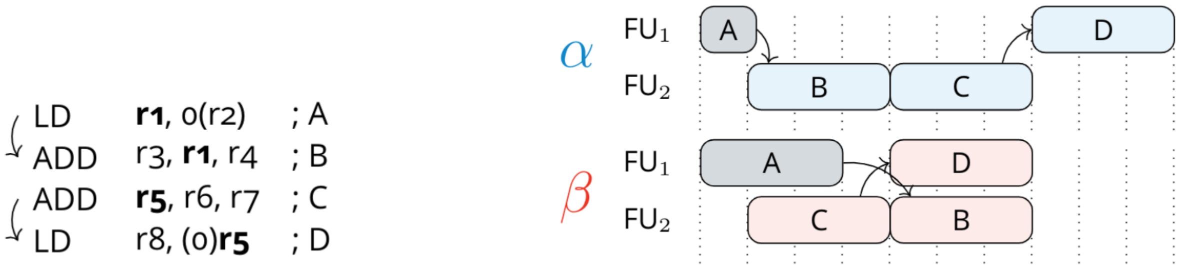
TODO: **stages of WCET-analysis**

2.5 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 2.3 shows the example of such an anomaly. Here, the assembly sequence consists of 4 instructions* (A,B,C,D) *with data dependencies* $A \to B$ *and* $C \to D$*. Figure 2.3b 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.*



```
LD    r1, 0(r2)    ; A
ADD   r3, r1, r4   ; B
ADD   r5, r6, r7   ; C
LD    r8, (0)r5    ; D
```

(a) Inpus assembly sequence

(b) Scheduling on functional units comparison

Figure 2.3: TA caused by variation in latency of instruction A (from [1])

non-composable architectures
amplification and counter-intuitive TAs

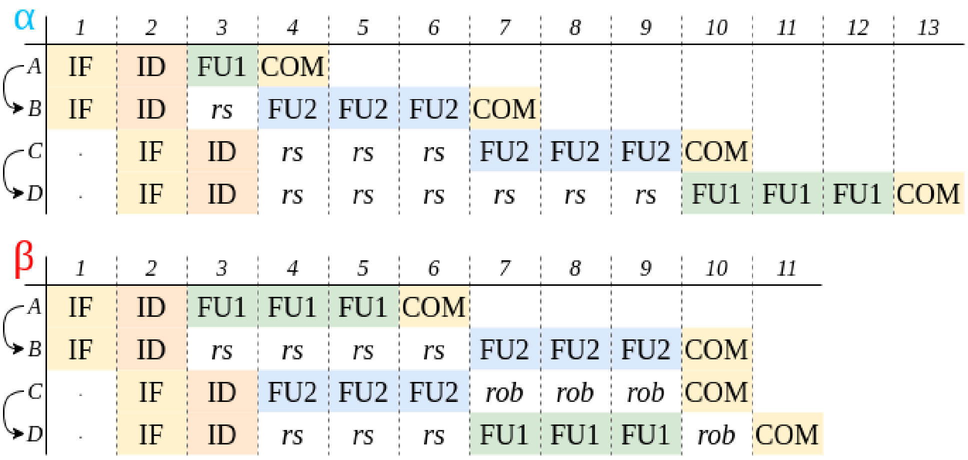
2.6 Execution diagrams

TODO: **what is trace, what is variation**

TODO: **vertical diagram**

TODO: **diagonal diagram**

TODO: **execution trace vs instruction trace distinction**



α	1	2	3	4	5	6	7	8	9	10	11	12	13
A	IF	ID	FU1	COM									
B	IF	ID	*rs*	FU2	FU2	FU2	COM						
C	·	IF	ID	*rs*	*rs*	*rs*	FU2	FU2	FU2	COM			
D	·	IF	ID	*rs*	*rs*	*rs*	*rs*	*rs*	*rs*	FU1	FU1	FU1	COM

β	1	2	3	4	5	6	7	8	9	10	11
A	IF	ID	FU1	FU1	FU1	COM					
B	IF	ID	*rs*	*rs*	*rs*	*rs*	FU2	FU2	FU2	COM	
C	·	IF	ID	FU2	FU2	FU2	*rob*	*rob*	*rob*	COM	
D	·	IF	ID	*rs*	*rs*	*rs*	FU1	FU1	FU1	*rob*	COM

Figure 2.4: Execution traces from example 1

— 3 —

State-of-the-Art

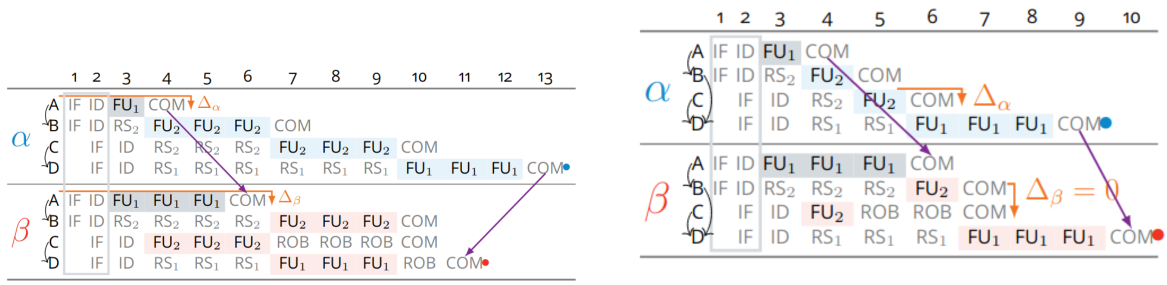
3.1 Evolution of TA-definitions

Several attempts were made to formally define the timing anomaly. Here we give a review of some definitions which can be applied to our architecture model.

3.1.1 Step Heights

Gebhard [3] gives a timing-anomaly definition based on local execution time of instruction in comparison to global execution time defined as sum of local ones. 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).



		1	2	3	4	5	6	7	8	9	10	11	12	13
α	A	IF	ID	FU_1	COM									
	B	IF	ID	RS_2	FU_2	FU_2	FU_2	COM						
	C		IF	ID	RS_2	RS_2	RS_2	FU_2	FU_2	FU_2	COM			
	D		IF	ID	RS_1	RS_1	RS_1	RS_1	RS_1	RS_1	FU_1	FU_1	FU_1	COM
β	A	IF	ID	FU_1	FU_1	FU_1	COM							
	B	IF	ID	RS_2	RS_2	RS_2	RS_2	FU_2	FU_2	FU_2	COM			
	C		IF	ID	FU_2	FU_2	FU_2	ROB	ROB	ROB	COM			
	D		IF	ID	RS_1	RS_1	RS_1	FU_1	FU_1	FU_1	ROB	COM		

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

		1	2	3	4	5	6	7	8	9	10
α	A	IF	ID	FU_1	COM						
	B	IF	ID	RS_2	FU_2	COM					
	C		IF	ID	RS_2	FU_2	COM				
	D		IF	ID	RS_1	RS_1	FU_1	FU_1	FU_1	COM	
β	A	IF	ID	FU_1	FU_1	FU_1	COM				
	B	IF	ID	RS_2	RS_2	RS_2	FU_2	COM			
	C		IF	ID	FU_2	ROB	ROB	COM			
	D		IF	ID	RS_1	RS_1	RS_1	FU_1	FU_1	FU_1	COM

(b) Counterexample to the definition

Figure 3.1: Gebhard's definition applied to execution traces (from [1])

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, the 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 complitions.

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.

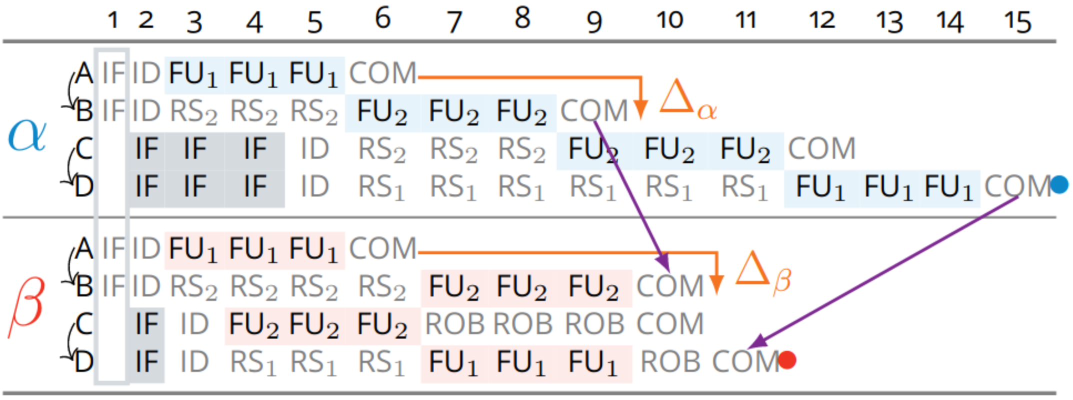

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
α
A IF ID FU_1 FU_1 FU_1 COM
B IF ID RS_2 RS_2 RS_2 FU_2 FU_2 FU_2 COM
C IF IF IF ID RS_2 RS_2 RS_2 FU_2 FU_2 FU_2 COM
D IF IF IF ID RS_1 RS_1 RS_1 RS_1 RS_1 RS_1 FU_1 FU_1 FU_1 COM
Δ_α
β
A IF ID FU_1 FU_1 FU_1 COM
B IF ID RS_2 RS_2 RS_2 RS_2 FU_2 FU_2 FU_2 COM
C IF ID FU_2 FU_2 FU_2 ROB ROB ROB COM
D IF ID RS_1 RS_1 RS_1 FU_1 FU_1 FU_1 ROB COM
Δ_β

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

This definition also leads to misleading effect where scenario [1] where it is clear that no time anomaly is present, but it is still detected by the definition. Figure 3.2 illustrates this contradiction. TODO: **details**

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.

3.1.4 Instruction Locality

3.1.5 Progress-based definition

Hahn and Reineke [5] intorduce 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: IF ↑, IF ↓, ID ↑, ID ↓, FU ↑, FU ↓ and *COM*. ↑ signs the acquisition of a resource and ↓ its release. *COM* denotes the acquisition of the commit stage, its release is not captured by the framework.

Latency is defined as a time difference between an acquisition of some resource and a release of it. For each pair of traces corresponding to the same program the sets of events are only different by the timestamps. Thus, for each event in one trace there is a corresponding event in the other one. Formally, it is a fucntion $CospEvent : Events(\alpha) \to Events(\beta)$.

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 accesss 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**

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

   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: $(I_1, \text{FU}\downarrow, t) \xrightarrow{0} (I_2, \text{FU}\uparrow, t')$.

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 $(I_1, FUr, t) \xrightarrow{0} (I_2, FUa, t')$.

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. $\Delta(e_\beta\downarrow, e_\beta) < \Delta(e_\alpha\downarrow, e_\alpha)$.

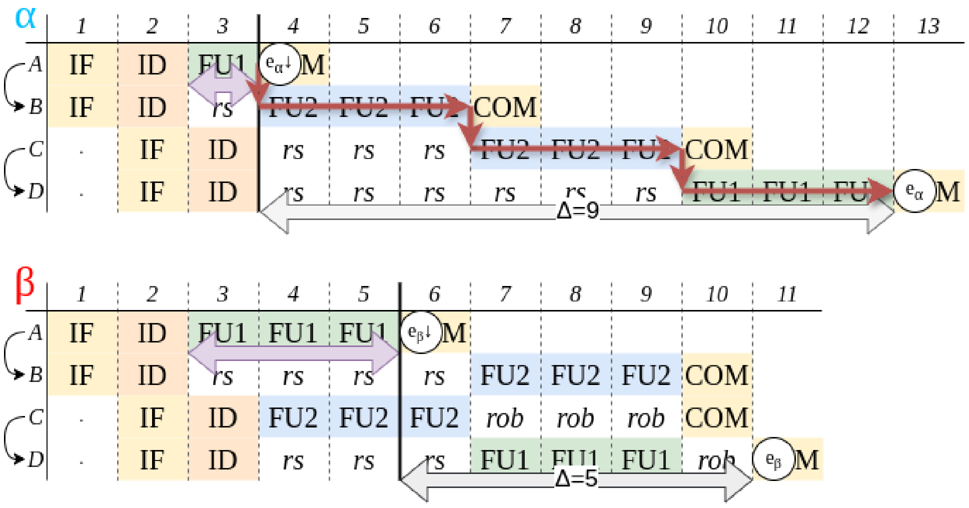

α
1 2 3 4 5 6 7 8 9 10 11 12 13
A IF ID FU1 eα↓ M
B IF ID rs FU2 FU2 FU2 COM
C · IF ID rs rs rs FU2 FU2 FU2 COM
D · IF ID rs rs rs rs rs rs FU1 FU1 FU1 eα M
Δ=9
β
1 2 3 4 5 6 7 8 9 10 11
A IF ID FU1 FU1 FU1 eβ↓ M
B IF ID rs rs rs rs FU2 FU2 FU2 COM
C · IF ID FU2 FU2 FU2 rob rob rob COM
D · IF ID rs rs rs FU1 FU1 FU1 rob eβ M
Δ=5

Figure 3.3: Causality-based TA detection applied for example 1. $e_\alpha\downarrow = (A, \text{FU}\downarrow, 4), e_\beta\downarrow = (A, \text{FU}\downarrow, 6), e_\alpha = (A, COM, 13), e_\beta = (A, COM, 11)$. 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.

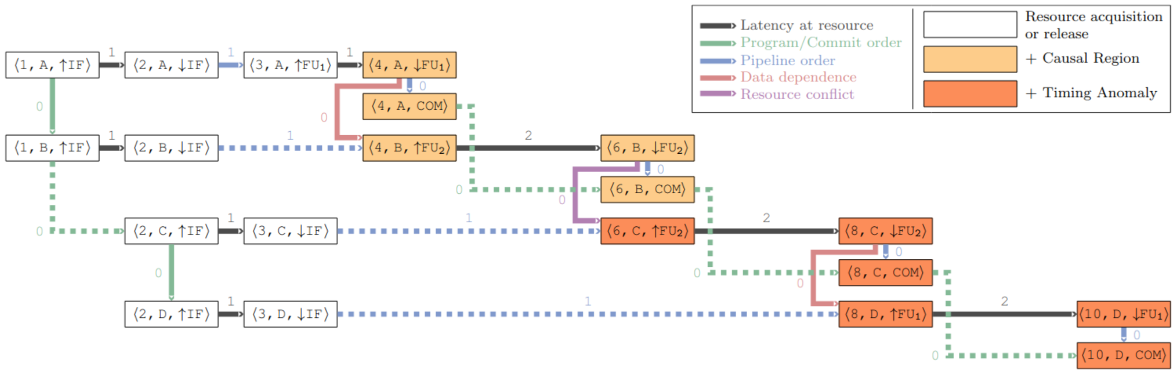


Figure 3.4: Complete ETDG for trace α from figure 2.4. TODO: **image source**

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

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.

TODO: **each pair of executions is considered? or all executions are compared agains the one referen**

4.1.1 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.

We can simplify this view by directly expressing the resource, dependencies and possible latencies of instruction.

give here an example

graph by python -> slow...

3 modes of using: manual, random, total

4.2 Adapting definition of Binder et al.

4.3 Gap problem

4.4 Formal Requirements for Causality Graph

4.5 New Causality Definition

4.6 Taking BP state into account

Put to conclusion?

4.7 Results

put examples of different TA here

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

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

Appendix

Appendix goes here...