Aalborg Universitet

What is a Timing Anomaly?

Cassez, Franck; Hansen, Rene Rydhof; Olesen, Mads Chr.

Published in:
Proceedings of the 12th International Workshop on Worst-Case Execution-Time Analysis

DOI (link to publication from Publisher):
10.4230/OASIcs.WCET.2012.1

Publication date:
2012

Document Version
Også kaldet Forlagets PDF

Link to publication from Aalborg University

Citation for published version (APA):
Cassez, F., Hansen, R. R., & Olesen, M. C. (2012). What is a Timing Anomaly? I T. Vardanega (red.), *Proceedings of the 12th International Workshop on Worst-Case Execution-Time Analysis* (Bind 23, s. 1-12). Schloss Dagstuhl. Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.WCET.2012.1

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: May 02, 2026

What is a Timing Anomaly?

Franck Cassez[1], René Rydhof Hansen*[2], and Mads Chr. Olesen[2]

1 **National ICT Australia**
Sydney, Australia
`Franck.Cassez@nicta.com.au`

2 **Department of Computer Science, Aalborg University**
Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark
`{rrh,mchro}@cs.aau.dk`

Abstract

Timing anomalies make worst-case execution time analysis much harder, because the analysis will have to consider all local choices. It has been widely recognised that certain hardware features are timing anomalous, while others are not. However, defining formally what a timing anomaly is, has been difficult.

We examine previous definitions of timing anomalies, and identify examples where they do not align with common observations. We then provide a definition for *consistently slower hardware traces* that can be used to define timing anomalies and aligns with common observations.

1998 ACM Subject Classification C.4 [Performance of systems]: Modelling techniques, Performance attributes

Keywords and phrases Timing anomalies, worst case execution time (WCET), abstractions

Digital Object Identifier 10.4230/OASIcs.WCET.2012.1

1 Introduction

Developing reliable real-time systems requires that guarantees on the run-time of tasks can be given, that hold under all circumstances i.e. regardless of the input data and previous execution history of the system. Typically the Worst-Case Execution Time (WCET) is the most important guarantee as it can be used to ensure the system responds in a timely manner.

However, modern processors are not optimized for worst cases, but optimize for improving the average case performance instead. This often makes their worst-case behaviour much harder to predict, and thus makes it harder to give absolute guarantees. One often hoped for property is that local worst-case timing choices will lead to the global worst-case timing — when this is not the case it is dubbed a *timing anomaly*. The classic example of a timing anomaly [6] is shown in Figure 1, where a cache miss for instruction A (bottom) is locally slower but turns out not to be the globally slowest (the top trace is slower). The example will be treated in greater detail later.

If an execution platform can be proven to be free of timing anomalies, very efficient techniques exist for analysing the worst-case timing behaviour [11]. On the contrary, if the execution platform exhibits timing anomalies there is little hope for using the same efficient abstraction techniques [6].

* Author partially funded by the “Certifiable Java for Embedded Systems” (CJ4ES) project (Danish Research Council for Technology and Production, grant number 10-083159).

© Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen;
licensed under Creative Commons License NC-ND
12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012).
Editor: Tullio Vardanega; pp. 1–12

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

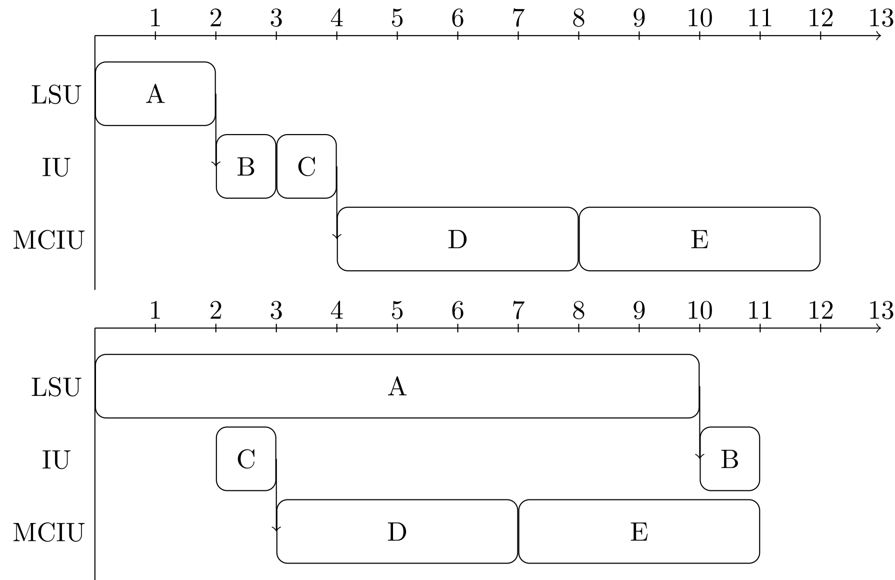

Figure 1 The canonical example of a timing anomaly from [6], where a cache miss (locally slower) leads to a scheduling that is globally faster. LSU, IU and MCIU are the three functional units that can execute out-of-order, but preference is given to older instructions.

Because of this, identifying timing anomalies has been an area of interest for some time, and some observations have been broadly recognised as being true:

- The LRU cache replacement policy is not timing anomalous.
- Other cache replacement policies such as FIFO and MRU exhibit timing anomalies [2, 4].
- In-order pipelines (without caches) are not timing anomalous.
- Resource allocation decisions (such as those presented by out-of-order execution or cache replacement) are a necessary condition for timing anomalies [10].

Using efficient abstraction techniques to compute the WCET is at the core of WCET analysis tools. However, the most powerful abstractions are sound only for timing anomaly free hardware. This explains why there have been some attempts to formally define timing anomalies [6, 9], but the various definitions have not been related to each other thus far.

In this work we will argue that the previous attempts are either too coarse or too precise to be used as universal definitions of timing anomalies. Each of the previous attempts definitely have their merits for application in connection with different analysis techniques (abstract interpretation, etc.), but a definition of timing anomalies should be as general as possible, while still retaining the property that the existence of timing anomalies forces the WCET analysis to consider more than one local choice.

Our Contribution

Our work is guided by the need for a definition of timing anomalies on the concrete model of the processor, instead of abstractions thereof. Consequently, in the following we propose a definition of timing anomalies that can be used in two different directions:

1. on hardware systems that are proven to be free of timing anomalies, the efficient abstraction techniques used in most WCET analysis tools are sound;

2. the definition we propose is based on the concrete reference hardware and only relates comparable hardware traces in order to avoid spurious timing anomalous diagnostics (see Section 4.2) resulting from abstraction of the hardware and/or of the hardware traces.

Without a definition of timing anomalies on the concrete reference hardware model, it is impossible to prove that abstraction is sound. Therefore we define timing anomalies as a property over different traces of the concrete hardware model.

But what traces should be comparable? We will argue that only traces resulting in the same instruction stream, i.e. the same program execution, should be comparable, in particular traces produced by different input data should not be comparable. It seems natural that different input data can result in different control flows, and therefore different instruction streams, where small changes in the input can result in much longer instruction streams, and therefore much longer execution times.

Another consideration is what elements of the hardware traces should be compared. Previous definitions have compared the timing of the first instruction with the timing of the last instruction in the stream [6], or made comparisons at points where the two traces have executed the same number of instructions [3]. We will argue that comparisons should be made on the *completion times* of each instruction.

Outline of the Paper

This work is divided into five sections: In Section 2 we define hardware systems and execution of programs on them. In Section 3 we define timing anomalies, and then examine related work in Section 4. We then compare the different definitions in Section 5, before concluding in Section 6.

2 Execution of Programs on Hardware

Before turning to timing anomalies, we first need to formalise our notion of hardware systems and how programs are executed on them. In order for our work to be applicable to a wide variety of systems, we aim to make as few assumptions about the hardware as possible. However, it usually consists of a processor and main memory (including caches). As usual, the hardware can process (machine code) instructions, taken from the set $\mathsf{Instructions}$, with each instruction located in memory at some address. We let $\mathsf{HardwareStates}$ be the (finite) set of possible hardware states and assume the hardware states contain the state of the memory.

The *semantics* of a hardware system is given by a transition system that specifies how the state of the hardware evolves in order to execute a program on given input data. We only model transitions between hardware states that take an observable amount of time and produce an observable result[1], e.g., finishing execution of a (set of) instruction(s).

The observable results, in the set $\mathsf{Observations}$, are not strictly necessary but are admitted as a convenience for later developments. In our work, the typical observations of interest in a given hardware system are the instructions that finish (in each cycle or time unit). We can now give the formal definition of a hardware system.

▶ **Definition 1** (Hardware System). A *hardware system* $\mathcal{H}$ is formalised as a *stutter-free* and *deterministic* labelled transition system $\mathcal{H} = (\mathsf{HardwareStates}, \mathsf{Time} \times \mathsf{Observations}, \rightarrow)$. The

[1] Bus latency, speculative execution, pipeline flushes, etc., are not visible and may generate extra cycles before an externally visible hardware state occurs.

transition relation $\rightarrow \subseteq \mathsf{HardwareStates} \times (\mathsf{Time} \times \mathsf{Observations}) \times \mathsf{HardwareStates}$ describes the *time* required to reach the next state and the externally visible *observations* produced by a transition.

As usual, a transition $(s, (t, o), s') \in \rightarrow$ is denoted $s \xrightarrow{(t,o)} s'$. The properties "stutter-free" and "deterministic" can then be formulated as follows: if $s \xrightarrow{(t,o)} s'$ then $s \neq s'$, and if $s \xrightarrow{(t,o)} s'$ and $s \xrightarrow{(t',o')} s''$ then $t = t', o = o'$ and $s' = s''$. A *run* in the hardware system $\mathcal{H}$ is defined to be a sequence $\sigma = s_0 \xrightarrow{(t_1,o_1)} s_1 \xrightarrow{(t_2,o_2)} \cdots \xrightarrow{(t_n,o_n)} s_n$ such that for all $1 \leq i \leq n-1$ it holds that $s_i \xrightarrow{(t_{i+1},o_{i+1})} s_{i+1}$ (in the $\mathcal{H}$ transition system) and the *length* of the run is defined as $length(\sigma) = n$. The *trace* of the run σ is $trace(\sigma) = (t_1, o_1) : (t_2, o_2) : \cdots : (t_n, o_n)$; the *time trace* of σ is $time(\sigma) = t_1 : \cdots : t_n$, and the *observation trace* of σ is $obs(\sigma) = o_1 : o_2 : \cdots : o_n$.

▶ **Example 2.** Observing the two traces shown in Figure 1 using "just finished instructions" as observations, we obtain the following run for the first (top) part of the example:

$$h_0 \xrightarrow{(2,\{A\})} h_1 \xrightarrow{(1,\{B\})} h_2 \xrightarrow{(1,\{C\})} h_3 \xrightarrow{(4,\{D\})} h_4 \xrightarrow{(4,\{E\})} h_5$$

and the run below for the second (lower) example in Figure 1:

$$h_0 \xrightarrow{(3,\{C\})} h_1 \xrightarrow{(4,\{D\})} h_2 \xrightarrow{(3,\{A\})} h_3 \xrightarrow{(1,\{B,E\})} h_4$$

Note that the observation on the final transition above shows that the two instructions labelled B and E finish simultaneously.

2.1 Execution of a Program on Hardware

We assume that all programs terminate. Given a program P and input data d in $\mathsf{Data}(P)$ (the set of admissible input data for P), the language semantics uniquely determine the *program trace*, i.e. the sequence of instructions to be performed to compute the result of program P on input d. In the following we need to be able to unambiguously identify specific occurrences of instructions in a program trace (the same instruction can be performed several times in the trace, for instance when loops are executed). Thus we formalise the program trace for (P, d) to be a mapping that assigns a unique index to each instruction in the program trace: $\mathsf{ProgramTrace}(P, d) : [1..k] \rightarrow \mathsf{Instructions}$ where k is the length of the trace and $\mathsf{ProgramTrace}(P, d)(j)$ gives the instruction executed at step j for each index $1 \leq j \leq k$.

▶ **Example 3.** The program trace for the example in Figure 1 is: $\mathsf{ProgramTrace}(P, d) = [1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E]$.

Given a hardware system $\mathcal{H}$, a program P and input data $d \in \mathsf{Data}(P)$, we let $I(P, d) \subseteq \mathsf{HardwareStates}$ be the hardware states that contain program P and input data d in memory, and where the first instruction of P is about to start execution. For $h_0 \in I(P, d)$, executing P with input d on $\mathcal{H}$ yields a unique sequence[2] of transitions in the hardware: $h_0 \xrightarrow{(t_1,o_1)} h_1 \cdots h_{n-1} \xrightarrow{(t_n,o_n)} h_n$. As the hardware is deterministic, each observation o_i can be taken to be a set of indices in $\{1, \ldots, k\}$: the indices uniquely identify the occurrences of instructions of $\mathsf{ProgramTrace}(P, d)$ being completed at each step. We can now formalise what it means to execute a program on a hardware system:

[2] Which we assume to be correct with regard to the instruction semantics.

▶ **Definition 4** ((P, d, h_0)-run)**.** Let $\mathcal{H}$ be a hardware system, P a program, d input data, and $h_0 \in \mathsf{HardwareStates}$. Then the run (in $\mathcal{H}$) $h_0 \xrightarrow{(t_1,o_1)} h_1 \cdots h_{n-1} \xrightarrow{(t_n,o_n)} h_n$ is called a (P, d, h_0)*-run* (in $\mathcal{H}$) whenever $h_0 \in I(P, d)$ and o_i is the set of (indices of) instructions completed during the transition $h_{i-1} \xrightarrow{(t_i,o_i)} h_i$ for $1 \leq i \leq n$.

▶ **Definition 5** (Completion Time)**.** Let $\mathsf{ProgramTrace}(P, d) : [1..k] \to \mathsf{Instructions}$ be the program trace of (P, d) and let σ be the corresponding (P, d, h)-run starting in $h \in I(P, d)$ with $trace(\sigma) = (t_1, o_1) : \cdots : (t_n, o_n)$. By $Ctime(\mathsf{ProgramTrace}(P, d)[j], h)$ we denote the *completion time* of instruction $1 \leq j \leq k$ finishing after transition m (i.e., $j \in o_m$) and define it as follows $Ctime(\mathsf{ProgramTrace}(P, d)[j], h) = \sum_{i=1}^{m} t_i$.

We let $Ctime(\mathsf{ProgramTrace}(P, d), h) = \max_{1 \leq j \leq k} Ctime(\mathsf{ProgramTrace}(P, d)[j], h)$ denote the maximal completion time for all instructions in the program and thus for completing the entire program.

▶ **Example 6.** The first trace in the example in Figure 1 has the following completion times for the 5 instructions: $[2, 3, 4, 8, 12]$ and for the second trace: $[10, 11, 3, 7, 11]$.

2.2 Exemplary Hardware Models

To be able to exemplify different phenomena we will use three different hardware models:

M_1 is a single-stage pipeline with a data-cache. The instructions of interest are the memory accesses, and the hardware model will be used to demonstrate timing anomalies with different cache replacement policies such as LRU and FIFO. For this reason we will simply denote instructions by the memory address they access.

M_2 is the simplified PowerPC architecture described in [6]. It is an out-of-order processor with three functional units: a Load/Store unit (LSU) communicating with a data cache, a Multi-Cycle Integer Unit (MCIU) and an Integer Unit (IU). For a detailed description see [6]. It is used for the classic example in Figure 1.

M_3 is a single-stage pipeline with no caches, but with a multiplication instruction `MUL` that takes 1 cycle if one of the operands is 0, and 2 cycles otherwise. This is a simplified version of the processor model in [3]. We extend M_3 with conditional execution of all instructions as on the ARM architecture [1].

3 Formalising Timing Anomalies

Slightly simplified, our notion of timing anomaly is based on the idea that timing anomalies only occur when a program is executed on a hardware system where no initial state gives rise to worse (slower) execution time than all other initial hardware states (modulo "irrelevant" parts of the hardware state). This approach requires us to formalise what it means for one program execution to be slower than, or rather: consistently as slow as, another execution (of the same program on the same data):

▶ **Definition 7** (Consistently as slow)**.** Let P be a program with input data d and let $\mathsf{ProgramTrace}(P, d) : [1..k] \to \mathsf{Instructions}$ be the program trace of (P, d). Let $h, h' \in I(P, d)$ and σ (respectively σ') be a (P, d, h)-run (respectively (P, d, h')-run). Then h' is said to be *consistently as slow* as h, denoted $h \sqsubseteq_{time} h'$, if and only if

$$\forall 1 \leq j \leq k \colon Ctime(\mathsf{ProgramTrace}(P, d)[j], h) \leq Ctime(\mathsf{ProgramTrace}(P, d)[j], h')$$

Intuitively the above definition compares the execution time of all prefixes of a program trace and requires one to be consistently as slow as the other.

▶ **Example 8.** Consider the example in Figure 1, where h is the hardware state resulting in the top run, and h' the state resulting in the bottom run.

$$Ctime(\mathsf{ProgramTrace}(P,d)[1],h) = 2 \leq Ctime(\mathsf{ProgramTrace}(P,d)[1],h') = 10$$

meaning instruction A was slower in the bottom trace, but

$$Ctime(\mathsf{ProgramTrace}(P,d)[5],h) = 12 \not\leq Ctime(\mathsf{ProgramTrace}(P,d)[5],h') = 11$$

meaning instruction E was not slower in the top trace, thus $h \not\sqsubseteq_{time} h'$. However instruction A in the bottom trace is still slower than in the top trace, so $h' \not\sqsubseteq_{time} h$.

The "consistently as slow" ordering is a pre-order (see below). However, it is not a partial order since two hardware states, which differ only in parts that are irrelevant to a given program, will still give rise to identical instruction completion times:

▶ **Lemma 9.** *For all programs P and input data d the relation $\sqsubseteq_{time}$ is a pre-order on $I(P,d)$.*

We can now propose a formal definition for timing anomalies:

▶ **Definition 10** (Timing Anomaly Free)**.** A hardware system, $\mathcal{H}$, is said to be *free of timing anomalies* with respect to program P and input d, if and only if there exists a maximal element, $\mathcal{W} \in I(P,d)$: $\forall h \in I(P,d)\colon h \sqsubseteq_{time} \mathcal{W}$.

Note that the maximal element is not necessarily unique: consider the case of LRU caches with no useful elements in them, hence all references resulting in cache misses.

The following lemma characterises (the absence of) timing anomalies in terms of upper bounds for arbitrary pairs of states. As shown in Section 5, this characterisation can be convenient when proving the presence of timing anomalies.

▶ **Lemma 11.** *A hardware system $\mathcal{H}$ is free of timing anomalies with respect to program P and input data d if and only if $\forall h, h' \in I(P,d)\colon \exists h'' \in I(P,d)\colon h \sqsubseteq_{time} h'' \wedge h' \sqsubseteq_{time} h''$.*

Proof. The "only if" direction is trivial and the "if" direction is proved by induction in the size of $I(P,d)$. ◀

Note that this does not require all hardware states to be ordered under $\sqsubseteq_{time}$, but only requires that for any pair of states a third state exists that gives rise to a consistently as slow run as both; i.e. it should be an upper bound for the pair of states.

Consider the example in Figure 2 where two runs of a LRU cache are not ordered either way, but we would still like to characterise LRU as not timing anomalous; there exists a consistently slower initial state than both, namely the empty cache.

Having defined timing anomaly free-ness for a given program and input data, it is straightforward to generalise the definition to cover entire hardware systems:

▶ **Definition 12** (Timing Anomaly Free Hardware System)**.** A hardware system, $\mathcal{H}$, is said to be *free of timing anomalies* for program P if and only if it is timing anomaly free for each $d \in \mathsf{Data}(P)$. Hardware $\mathcal{H}$ if free of timing anomalies if and only if it is timing anomaly free for all programs P (valid for $\mathcal{H}$).

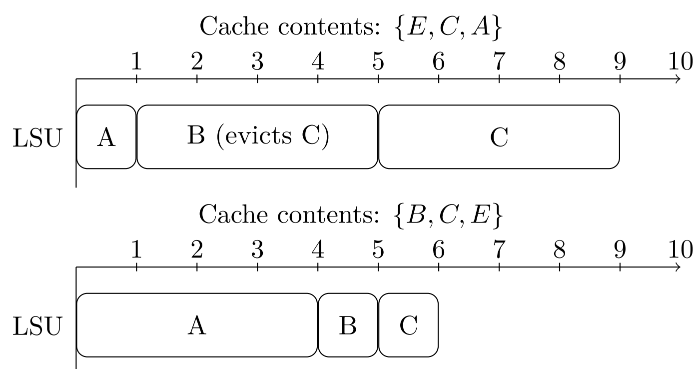

Figure 2 Two runs of the program `LD A; LD B; LD C` on hardware model M_1 with a LRU cache. The cache contents are ordered sets of data elements, from newest to oldest.

Finally we relate our definition of "consistently as slow as" to the definition of the WCET for a program P on hardware $\mathcal{H}$.

▶ **Definition 13** (Worst Case Execution Time (WCET)). The *worst case execution time* for a program P (on $\mathcal{H}$) is defined as follows:

$$WCET_{\mathcal{H}}(P) = \max_{h \in I(P,d), d \in \mathsf{Data}(P)} \{Ctime(\mathsf{ProgramTrace}(P,d), h)\}$$

If $\mathcal{H}$ is free of timing anomalies for P, only a maximal element in $I(P,d)$ need be considered. Indeed, if $h \sqsubseteq_{time} h'$, then $Ctime(\mathsf{ProgramTrace}(P,d), h) \leq Ctime(\mathsf{ProgramTrace}(P,d), h')$. Definition 13 is then reduced to computing $\max_{d \in \mathsf{Data}(P)}\{Ctime(\mathsf{ProgramTrace}(P,d), h) | h \text{ maximal in } I(P,d)\}$.

4 Related Work

4.1 Defining Timing Anomalies by Changes in Instruction Latency

Timing anomalies were first discovered by Lundqvist and Stenström in [6, 5]. Their definition is re-used in [10], and we formulate it here in our framework.

Assume a sequence of instructions $\pi = i_1 : \cdots : i_n$, with corresponding latencies $\tau_\pi(i_j)$, and total execution time C. Consider a situation where there exists a latency variation, Δt, such that the same sequence of instructions, but with a modified latency for the first instruction $\tau'_\pi(i_1) = \tau_\pi(i_1) + \Delta t$, results in a different sequence of instruction latencies $\tau_\pi(i_1) + \Delta t : \tau_\pi(i_2) : \cdots : \tau_\pi(i_n)$ and thus a possible different total execution time C', and thus a timing difference of $\Delta C = C' - C$.

▶ **Definition 14** (Timing Anomalies by Changes in Instruction Latency [10]). A timing anomaly is defined as a situation where according to the sign of Δt one of the following cases become true:

a) Increase of the latency: $\Delta t > 0 \implies (\Delta C > \Delta t) \vee (\Delta C < 0)$
b) Decrease of the latency: $\Delta t < 0 \implies (\Delta C < \Delta t) \vee (\Delta C > 0)$

This definition has some drawbacks:

- As pointed out in [9] there is an underlying assumption that the latency change of the first instruction does not influence the latencies of the subsequent instructions. This is not always the case.

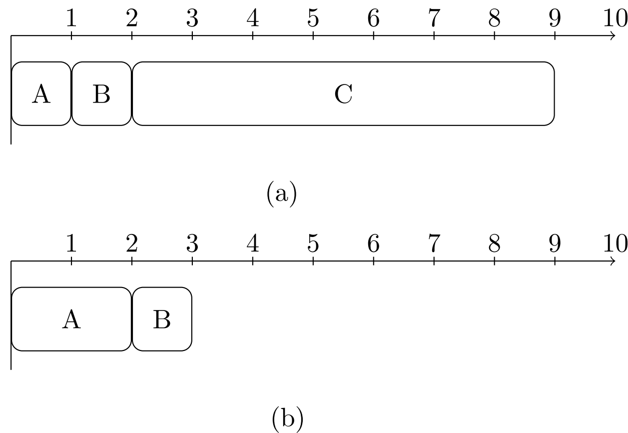

Figure 3 Example program run on M_3. The program is `A: MUL` R_0, R_0, R_1; `B: BRZ` R_0, C, where C is a linear, data-independent, subprogram summarised into one instruction. The instruction `BRZ` is interpreted as "branch to C if R_0 is zero". The two traces are (a) where $R_0 = 0$, and (b) where $R_0 = 1$.

- In [6] the change in latency can be unrelated to a change in hardware state, resulting in the definition deeming a hardware platform to suffer from timing anomalies, while the actual platform does not.

In [10] the second point is alleviated as the change in latency is assumed to be associated to two different initial hardware states, which are further assumed to be "almost identical". However without a formalisation of "almost identical" it could be argued that the two LRU caches in Figure 2 are almost identical, and thus would be deemed timing anomalous.

4.2 Defining Timing Anomalies by Abstract Models

In [9] a formal definition of timing anomalies is given. It however states that "Non-determinism – which is necessary for timing anomalies – is only introduced by abstraction", and goes on to define timing anomalies in terms of non-deterministic hardware models. We note that timing anomalies as demonstrated originally in [6] do not involve non-determinism, but instead involve concrete traces run on the same concrete (deterministic) hardware model.

We will argue that non-determinism is *not* necessary for timing anomalies to occur. Indeed, there is a strong link between the presence of timing anomalies and the existence of a sound deterministic over-approximating model of the timing of the hardware, but the causality is not both ways. In some cases timing anomalies can even occur as artefacts of the abstraction, even though they are not present in the concrete hardware.

As an example consider the two traces in Figure 3. Here different input data results in very different instruction streams, sharing similarities with timing anomalous traces. In our opinion this should not be characterised as a timing anomaly, as this would render practically all programs accepting input on all platforms to be timing anomalous. The definition in [9] requires that the two traces must have the same instruction streams, and thus Figure 3 is not a timing anomaly by that definition.

We however note that the same instruction stream can still result in two very different timing behaviours, when given different input data. As an example, the ARM architecture allows the conditional execution of most instructions. We can thus derive an example where the same instruction stream gives rise to timing anomalous behaviour on different input data,

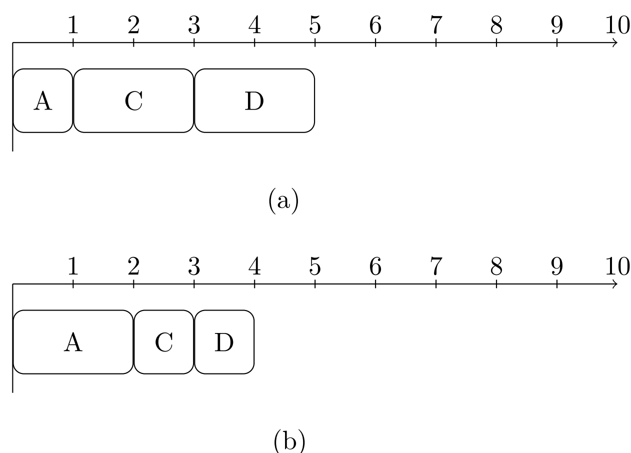

Figure 4 The example from Figure 3, but instead of branching it uses conditional execution. Therefore the two instruction streams are the same, but the processor treats C as a no-op in (b). The program is `A: MUL` R_0, R_0, R_1; `C: MULNZ` R_2, R_2, R_1; `D: MULNZ` R_2, R_2, R_1 , where we let A set the condition flags. Thus C and D only gets executed if R_0 is not 0.

as seen in Figure 4.

According to the definition in [9] this would be timing anomalous, as there exists a non-local worst-case path (the A instruction in (a)) resulting in a globally longer path, than all local worst-case paths (b).

In [8] a slightly relaxed definition from [9] is given, which is used to compute upperbounds on the the possible error in WCET estimation between two hardware states. However, with regards to the examples we consider there is no difference.

In the same line [3] describes a method to identify timing anomalies in a processor using bounded model checking. This is done by comparing the execution time of the same instruction stream on two different processors: the real processor, and an (abstract) "always-worst case" performing processor. If the "worst-case" processor can overtake the real processor, the processor is deemed to have timing anomalies.

However [3] uses abstraction of input data and thus ends up comparing execution traces which can only result from different input data: The trace given in [3] (a) cannot occur on the real processor with the same input data as the trace in (b). For trace (a) to occur one of the operands (R_4 and R_6 in this case) needs to be 0, that is $R_4 = 0 \vee R_6 = 0$. However for trace (b) to occur none of the operands can be 0, thus $R_4 \neq 0 \wedge R_6 \neq 0$. As these two conditions are the negation of one another, the two traces cannot occur with the same input data. Since the two traces cannot occur with the same input data, they will be incomparable, per our Definition 7. Of course, hardware can be viewed abstractly. For timing analysis it is very important that these abstractions are sound, i.e. overapproximating the timing.

▶ **Definition 15** (More Favorable Hardware). Hardware $\mathcal{H}$ with hardware states HardwareStates is more favorable than hardware $\mathcal{H}'$ with hardware states HardwareStates′, written $\mathcal{H} \sqsubseteq \mathcal{H}'$, if there exists a mapping α : HardwareStates $\rightarrow$ HardwareStates′ $\forall P, \forall d \in \mathsf{Data}(P) : h \sqsubseteq_{time} \alpha(h)$, where $\sqsubseteq_{time}$ is extended across different hardware systems.

Typically α is an abstraction function. Clearly, if $\mathcal{H} \sqsubseteq \mathcal{H}'$, then for any program P, $WCET_{\mathcal{H}}(P) \leq WCET_{\mathcal{H}'}(P)$. $\mathcal{H}'$ is thus a sound abstraction for computing the WCET of any program. The technique presented in [3] is a very valuable tool in finding sound

abstractions, however, the unsoundness of an abstraction cannot be translated into the real hardware exhibiting timing anomalies.

5 Results

We will now compare the different definitions of timing anomalies, and how they hold for and apply to different examples:

	Our Definitions 10, 12	Latency change [6, 10]	Abstraction [3, 9, 8]
Classic Ex. [6] (Fig. 1)	Yes, Lemma 16	Yes	Yes
LRU Cache	No, Lemma 19	Inapplicable1)	No
FIFO Cache	Yes, Lemma 20	Inapplicable1)	Yes4)
Branching (Fig. 3)	No, Lemma 17	Inapplicable2)	No4)
Conditional exec. (Fig. 4)	No, Lemma 18	Yes	Yes4)
MUL 0-speedup [3, Fig. 3]	No	Yes3)	Yes4)

1) Because the latency change can in general not be limited to, or contained within, the first instruction.
2) Because the sequence of instructions are not the same.
3) If we allow the latency change to occur on the second instruction.
4) Depends on the abstraction.

▶ **Lemma 16.** *The classic example in Figure 1 is timing anomalous by Definition 10.*

Proof. We will show that none of the two initial hardware states is consistently worse than the other, per Definition 7, and thus no upper bound can exist, per Definition 10. The only two initial hardware states that are relevant to consider is the cache where the data item referenced by A is in the cache, and a state where the data item referenced by A is not in the cache. Since there are only two initial hardware states, one or both of them would have to be consistently slower than the other. In Example 8 we already showed that none of the two traces is consistently slower than the other. Therefore, Definition 10 cannot be fulfilled. ◀

▶ **Lemma 17.** *The control-flow example in Figure 3 is not timing anomalous by Definition 10.*

Proof. Since M_3 has no cache, there is actually only one initial hardware state, h_0, where the first instruction is able to enter the processor in the first cycle: the empty pipeline. For every program P and data d there is therefore only one element in $I(P, d)$. By Definition 10 and the reflexivity of $\sqsubseteq_{\text{time}}$ the hardware system is timing anomaly free. ◀

▶ **Lemma 18.** *The "branching by conditional execution" example in Figure 4 is not timing anomalous by Definition 10.*

Proof. The argumentation is the same as for Lemma 17. ◀

▶ **Lemma 19.** *LRU caches are not timing anomalous by our Definition 12.*

Proof. A stronger statement can actually be proven: that the empty cache is always the worst initial hardware state for LRU caches [7]. By Definition 10 this satisfies Definition 12. ◀

▶ **Lemma 20.** *FIFO caches are timing anomalous by our Definition 12.*

Proof. Consider the two traces in Figure 5, none of which are consistently slower than the other. By Lemma 11 an upper bound should exist. An upper bound would have to have misses for all accesses. By enumeration of all distinct initial caches, none of them have misses for all accesses, and thus no upper bound for the two traces exist. ◀

	d e		b a	
a	a d	x	b a	
c	c a	x	c b	x
a	c a		a c	x
b	b c	x	b a	x
c	b c		c b	x
a	a b	x	a c	x
b	a b		b a	x
c	c a	x	c b	x

Figure 5 Example of a timing anomaly for the FIFO cache on M_1, adopted from [2]. The first line is the initial state of the cache for the two traces. The first column is the access sequence, and the x'es indicate cache misses.

6 Conclusion and Future Work

In this work we have looked at previous definitions of timing anomalies, and identified flaws in them. Specifically in their applicability to various types of known timing anomalies, but also in what examples they deem to be timing anomalies. We have proposed a definition of timing anomalies in terms of the existence of a consistently worst initial hardware state, in the concrete model of the hardware and shown that it coincides with common knowledge about timing anomalies.

The next step is to provide an operational definition of timing anomalies that enables us to effectively check whether some hardware is timing anomalous, and if it is, identify a set of initial hardware states, such that they are consistently worse than all other hardware states. This would enable a WCET analysis by simulating the execution of these initial states. The framework we have proposed can also be used to take advantage of the efficient abstraction techniques to over-approximate WCET on timing anomalous platforms: given $\mathcal{H}$ which is timing anomalous, define $\mathcal{H}'$ that soundly approximates $\mathcal{H}$ and show that $\mathcal{H}'$ is timing anomaly free.

References

1. *ARM920T Technical Reference Manual*, 1 edition, 2001.
2. C. Berg. PLRU Cache Domino Effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.
3. J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, pages 13–18, 2006.
4. G. Gebhard. Timing Anomalies Reloaded. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15, pages 1–10, 2010.
5. T. Lundqvist. *A WCET analysis method for pipelined microprocessors with cache memories*. PhD thesis, Chalmers University of Technology, 2002.
6. T. Lundqvist and P. Stenstrom. Timing Anomalies in Dynamically Scheduled Microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.
7. J. Reineke and D. Grund. Sensitivity of Cache Replacement Policies. Technical Report 36, March 2008. ISSN: 1860-9821, http://www.avacs.org/.

8 J. Reineke and R. Sen. Sound and Efficient WCET Analysis in the Presence of Timing Anomalies. In *9TH INTERNATIONAL WORKSHOP ON WORST-CASE EXECUTION TIME ANALYSIS*, page 101, 2009.

9 J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

10 I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*, pages 295–303, 2005.

11 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.