View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Timing Anomalies Reloaded

Gernot Gebhard[1]

1 **AbsInt Angewandte Informatik GmbH**
Science Park 1, D-66123 Saarbrücken, Germany
gebhard@asbint.com

Abstract

Computing tight WCET bounds in the presence of timing anomalies – found in almost any modern hardware architecture – is a major challenge of timing analysis.

In this paper, we renew the discussion about timing anomalies, demonstrating that even simple hardware architectures are prone to timing anomalies. We furthermore complete the list of timing-anomalous cache replacement policies, proving that the most-recently-used replacement policy (MRU) also exhibits a domino effect.

1998 ACM Subject Classification B.2.2

Keywords and phrases Timing Anomalies, Domino Effects, MRU Replacement Policy, LEON2

Digital Object Identifier 10.4230/OASIcs.WCET.2010.1

1 Introduction

The validation of the timing behavior of tasks in a safety-critical embedded software system requires both safe and precise worst case execution time (WCET) bounds. Those bounds need to be safe to ensure that each component of the software system performs its job in time. Furthermore, those bounds are required to be precise to ensure the schedulability of such software systems. Two different approaches have emerged to solve the timing analysis problem: measurement-based timing analysis and static WCET analysis. In the following, we focus on static timing analysis and one of the main challenges this analysis method has to face: timing anomalies.

Intuitively spoken, a timing anomaly is a counterintuitive behavior of a hardware architecture, where a "good" event (e.g., a cache hit) leads to an overall longer execution, whereas the opposing "worse" event, such as a cache miss, leads to a globally shorter execution time. In the presence of such anomalies, the local worst case is not always a safe assumption in static timing analysis. To compute safe timing guarantees, any static timing analysis has to consider all possible executions caused by any non-determinism in the abstract hardware model (e.g., such as unknown cache contents). Due to the loss of predictability, the static analysis of architectures featuring timing anomalies requires much more effort in terms of computational power and memory consumption.

Intuitively, one would assume that timing anomalies are restricted to complex hardware architectures. In fact, the Motorola PowerPC 755 is known to have a timing anomaly due to its complex pipeline [10]. Furthermore, architectures with caches with, e.g., PLRU or random replacement policies feature timing anomalies as well [2].

However, even simple architectures can suffer from timing anomalies, as demonstrated throughout this paper. We demonstrate that the LEON2 processor, developed at Aeroflex Gaisler [1], also features a timing anomaly caused by the processor's cache line fill mechanism.

In addition to discussing the LEON2, we complete the list of commonly used replacement policies that are prone to timing anomalies by examining the MRU replacement policy.

Section 2 discusses related work. Section 3 formally defines timing anomalies and introduces the existing timing anomaly classifications. Section 4 discusses the MRU replacement policy and

© Gernot Gebhard;
licensed under Creative Commons License NC-ND
10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010).
Editor: Björn Lisper; pp. 1–10

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

proves that this replacement strategy exhibits a timing anomaly. Section 5 introduces the LEON2 architecture and demonstrates that this architecture is prone to a timing anomaly – despite its rather simple structure. Finally, Section 6 concludes this paper.

2 Related Work

Lundqvist and Stenström [7] are the first to introduce the term timing anomaly. They find that the worst case instruction execution time behavior does not necessarily contribute to the global worst case execution time. In their paper the authors provide an example of a timing anomaly, where a cache hit leads to the worst case timing. Engblom and Jonsson [5] also discuss timing anomalies. They translate the notion of timing anomaly of Lundqvist and Stenström [7] to their model considering (local) timing of pipeline stages instead of whole instructions.

Both Lundqvist and Engblom claim that timing anomalies cannot occur in processors that only comprise in-order resources (i.e., two instructions can only use a resource in program order). This statement is unfortunately not always true, as we show by means of the LEON2 hardware architecture.

Schneider [10] describes a timing anomaly present in the Motorola PowerPC 755 (MPC755). The possibility to dispatch an instruction on two execution units with different behavior in conjunction with pipeline stalls triggers the described timing anomaly.

Thesing [11] discusses the Motorola ColdFire 5307 that has a rather simple in-order pipeline. He shows that the processor exhibits timing anomalies, caused by the pseudo round-robin cache replacement algorithm.

Berg [2] discusses cache replacement policies and their timing anomalies. He finds that caches using first-in first-out (FIFO), round-robin, or pseudo least-recently-used (PLRU) cache replacement strategies suffer from timing anomalies. These replacement strategies are commonly used in embedded hardware architectures, as they require less update logic compared to the LRU policy, which is free of timing anomalies.

In the context of WCET analysis, Reineke et al. [9] provide the first formal definition of timing anomalies. The paper provides a classification of timing anomalies, which we adopt in this paper.

Eisinger et al. [4] provide a novel methodology to automatically detect timing anomalies. Requiring an accurate hardware model to be available (e.g., in VHDL), the approach computes an instruction sequence that triggers a timing anomaly, if such a sequence exists. Yet, the approach is not fully automatic, because hardware features potentially causing timing anomalies need to be identified manually.

Reineke and Sen [8] discuss a related approach. Other than Eisinger et al., they propose a method that allows a static timing analysis to safely discard analysis states by means of Δ functions. A Δ function computes the maximal difference in timing between two system states on any input instruction sequence. For any pair of system states, a static timing analysis can consult the corresponding Δ function to determine which of the two states can be safely discarded. This works well for small hardware models, as demonstrated in the paper, but it remains unclear whether this approach can be applied to complex embedded architectures in a beneficial way.

Kirner et al. [6] show that splitting up a WCET analysis into separate parallel WCET analyses (corresponding to hardware components operating in parallel) is not generally safe in the presence of timing anomalies. Furthermore, the authors identify special instances of "parallel' timing anomalies still making a parallel decomposition of the WCET problem feasible. Their findings correspond to the classification of architectures (see Section 3.6). Non-fully timing compositional architectures do not allow for a safe, parallel decomposition of the WCET problem.

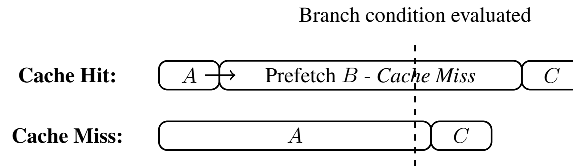

Figure 1 *Speculation-triggered timing anomaly*: The processor executes a conditional branch instruction, whose condition is yet unresolved. Assuming a cache hit for the initial code fetch, the processor speculatively fetches the instruction B that is not contained in the cache. This causes an overall longer execution time, because the cache line fill operation stalls the processor longer than it takes to resolve the branch condition.

3 Timing Anomalies

Intuitively, a timing anomaly is a counterintuitive behavior of a hardware architecture, where a local speed-up leads to a global slow-down. Several – average-performance increasing – hardware features may exhibit this kind of non-local execution time behavior. In the following we formally define timing anomalies in accordance to the definition found in [9]. Furthermore, we discuss some examples of timing anomalies commonly found in modern processors.

3.1 Formal Definition

▶ **Definition 1.** **(Hardware State)** *For a given hardware architecture $\mathcal{A}$, the set $\widehat{\mathcal{H}}$ comprises all possible* hardware states *of that architecture. A specific* hardware state *of the architecture is $\eta \in \widehat{\mathcal{H}}$.*

▶ **Definition 2.** **(Program)** *A* program *P is a directed graph $P = (V, E)$ with $E \subseteq V \times V$, where the nodes V represent instructions, and an edge $(u, v) \in E$ represents the control flow transition from instruction u to v. A* path *$\vec{\pi} = \pi_1 \pi_2 \dots$ through the program $P = (V, E)$ is a possibly infinite sequence of instructions, such that $(\pi_i, \pi_{i+1}) \in E$ for each $i < |\vec{\pi}|$.*

▶ **Definition 3.** **(Execution)** *The* execution *$\gamma_{\vec{\pi}}$ of the path $\vec{\pi}$ is a function $\widehat{\mathcal{H}} \times \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ assigning each instruction on the path $\vec{\pi}$ a non-negative (relative) execution time depending on the initial hardware state η (i.e, $\gamma_{\vec{\pi}}(\eta, i)$ denotes the time the processor needs to execute the instruction π_i). The (absolute)* execution time *under the initial state η until the position $n \in \mathbb{N}$ is $\Gamma_{\vec{\pi}}(\eta, n) = \sum_{i=1}^{n} \gamma_{\vec{\pi}}(\eta, i)$.*

▶ **Definition 4.** **(WCET)** *The* worst case execution time (WCET) *is determined by the worst-case initial hardware state θ, such that $\lim_{n \to |\vec{\pi}|} \Gamma_{\vec{\pi}}(\eta, n) \leq \lim_{n \to |\vec{\pi}|} \Gamma_{\vec{\pi}}(\theta, n)$ for all hardware states η.*

Note that Definition 3 allows for different initial hardware states before the execution of the program. This makes sense, because the precise initial state of the processor is usually unknown in the scope of static timing analysis. Thus static timing analyses need to consider all possible initial hardware states to provide safe WCET bounds.

▶ **Definition 5.** **(Timing Anomaly)** *An architecture has a* timing anomaly *if there exists a path $\vec{\pi}$ through a program P, $i, n \in \mathbb{N}$ with $n > i$, and a hardware state θ, such that $\gamma_{\vec{\pi}}(\theta, i) < \gamma_{\vec{\pi}}(\eta, i)$ and $\Gamma_{\vec{\pi}}(\theta, n) \geq \Gamma_{\vec{\pi}}(\eta, n)$ for all hardware states $\eta \neq \theta$. The state θ is called* timing-anomalous.

3.2 Examples

Figure 1 gives an example of a timing anomaly caused by the interaction between the branch prediction mechanism, the instruction cache, and the processor's ability to execute instructions out-of-order. In

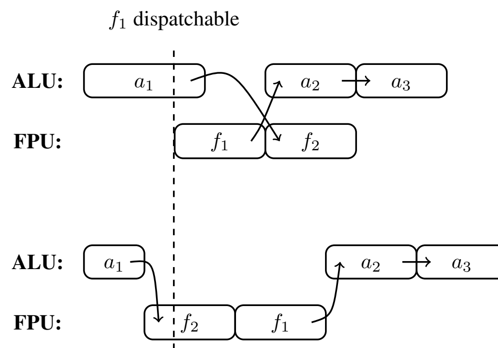

Figure 2 *Variant execution time triggered timing anomaly*: This example demonstrates that a locally fast instruction execution might cause a global slow-down of an instruction sequence. The instructions a_1, a_2, and a_3 execute on the ALU. The ALU features an early-out mechanism that allows integer divide instructions, such as a_1, to complete faster under certain circumstances. The other instructions f_1 and f_2 solely execute on the FPU. Edges between instructions indicate definition-use dependencies.

this example the processor is currently executing a conditional change-of-flow instruction, whose condition is not yet evaluated at the moment the instruction A is about to be fetched. Upon a cache hit for code fetch of instruction A, the processor starts to speculatively fetch the uncached branch target B. Although the initial cache hit locally causes a faster execution, the overall execution is slowed down, because the cache line fill fetching B takes longer than resolving the branch condition.

This timing anomaly could also be caused by speculative execution. This means that the processor starts to execute the fetched instructions, while the processor computes the branch condition. Instead of fetching the instruction B and being stalled due to a cache miss, the processor could speculatively execute the previously fetched instruction B resulting in a longer stall of the processor's pipeline.

Figure 2 demonstrates a different timing anomaly caused by instructions with variant execution times (e.g., due to dividers with an early-out mechanism). Here, the processor features two execution units, an arithmetical logical unit (ALU) and a floating point unit (FPU). Depending on the input parameters, the ALU executes integer division instructions, like a_1, quicker. In this case, completing instruction a_1 earlier, the processor is able to dispatch instruction f_2 in front of instruction f_1. This effectively causes the processor to execute all instructions sequentially. The instruction sequence takes longer to complete, because the processor cannot benefit from its ability to execute instructions in parallel. On the contrary, if instruction a_1 takes longer to complete, the processor will dispatch instruction f_1 earlier. This allows the processor to execute the instructions a_2 and f_2 in parallel, resulting in an overall faster execution.

The variable-execution-time-triggered timing anomaly corresponds to a so-called scheduling anomaly. In the same fashion, a task that terminates earlier could lead to an overall longer schedule. Whereas a faster schedule could be achieved if the very same task would run to completion a bit later. Greedy schedulers are usually unable to prevent this kind of anomaly.

3.3 Domino Effects

The presence of timing anomalies increases the complexity of static timing analyses. A static timing analysis cannot always assume the local worst case, if the analyzed architecture is prone to a timing analysis. Instead the analysis has to take all possibilities into account to compute safe WCET bounds.

Often the effect of a timing anomaly on the execution time stabilizes eventually. This means that any timing-anomalous execution reaches a point where it only differs by a constant from any other

execution on the sequence of the input program. Such a timing anomaly is called *k-bounded timing anomaly*, where k is the maximal difference in execution time caused by the timing anomaly. This is formalized in Definition 6. In the presence of a k-bounded timing anomaly, a static timing analysis could always assume the local worst case, adding the constant k to the computed WCET bound [8].[1]

► **Definition 6.** **(k-bounded Timing Anomaly)** *An architecture has a k-bounded timing anomaly, if there exists a $k \in \mathbb{R}_{\geq 0}$ such that for all timing-anomalous hardware states θ for the execution of a path $\vec{\pi}$ through a program P holds: $\Gamma_{\vec{\pi}}(\theta, n) - \Gamma_{\vec{\pi}}(\eta, n) \leq k$ for all $n \in \mathbb{N}$ and all states η.*

Unfortunately, some hardware features cause timing anomalies whose effects on timing are unbounded. Such timing anomalies are known as *domino effects*. Domino effects are essentially different from k-bounded timing anomalies: A k-bounded timing anomaly occurring in a loop only has a limited timing effect that eventually stabilizes. In other words, the loop body runtime will only differ for a bounded number of iterations and converge finally. In the presence of a domino effect, the loop body runtime will take different values without convergence in the future.

► **Definition 7.** **(Domino Effect)** *An architecture has a* domino effect*, if it exhibits a timing anomaly that is not k-bounded. Such timing anomalies are also known as* unbounded timing anomalies*.*

Domino effects are real. Schneider [10] has demonstrated that the MPC755 pipeline actually causes a domino effect. Furthermore, Berg [2] was able to show that, in contrast to the LRU replacement policy, the pseudo LRU, the FIFO, and the round-robin replacement strategies suffer from domino effects. Section 4 completes this list, proving the MRU policy to feature a domino effect as well.

3.4 Challenges for Static Timing Analysis

The presence of timing anomalies impacts both performance and precision of a static timing analysis. In general, an analysis must not always choose the locally most expensive execution, as this decision might not always lead to the global worst case execution time. Consequently, the number of states to consider during analysis time increases greatly, if the absence of timing anomalies cannot be proven for an analysis state, where multiple successor states are possible.

Furthermore, the inability of proving the absence of a timing anomaly might also lead to an increase in the computed WCET bound. The timing anomaly discussed in Section 5 can lead to an overestimation of up to 20% (strongly depending on the analyzed program).

3.5 Classification of Timing Anomalies

Reineke et al. [9] discern three different classes of timing anomalies:

- *Scheduling timing anomaly*: Most timing anomalies found in the literature correspond to this class of timing anomalies. Figure 2 is actually an instance of a scheduling timing anomaly. Depending on the execution time of a task, a faster execution might lead to a globally longer schedule. This kind of anomaly is well-known in the scheduling domain and has been thoroughly studied on various scheduling routines.
- *Speculation timing anomaly*: Figure 1 demonstrates such a timing anomaly. An initial cache hit (the local best case) causes a speculative prefetch addressing an instruction that is not cached. The cache miss leads to an overall longer execution. Section 5 discusses a speculation anomaly found in the LEON2 core.

[1] Note that k is an overestimation of the caused effect on timing. In most cases the precision of a static timing analysis will degrade by assuming the local worst case and adding the constant k to the computed WCET bound.

- *Cache timing anomaly*: Cache timing anomalies are caused by some non-LRU cache replacement strategies. Various cache replacement algorithms have been proven to cause domino effects.

The above classification sorts timing anomalies in accordance to the hardware property that is responsible for the timing anomaly. So far, this classification appears to be exhaustive in the sense that it covers all possible hardware features that might trigger timing anomalies.

Yet, the sole knowledge about the timing anomaly class does not suffice for static timing analysis. In addition to the hardware feature causing the anomaly, it is necessary to know the kind of anomaly. A static timing analysis of a hardware architecture that has a k-bounded timing anomaly can be realized with less effort[2] than a static analysis of an architecture that suffers under a domino effect as discussed in Section 3.3.

Currently, it is unclear how to determine whether an instance of a timing anomaly is k-bounded for a certain system state. Depending on the initial hardware state, a hardware feature triggering a domino effect might only cause a constantly-bounded anomaly for this special case. Furthermore, there exists no general approach to compute the constant k for a k-bounded timing anomaly.

3.6 Classification of Architectures

Depending on whether a hardware architecture exhibits k-bounded timing anomalies or domino effects, the architecture can be classified into three categories. Wilhelm et al. introduce the following categorization in [12]:

- *Fully timing compositional architectures:* The architecture does not exhibit any timing anomalies. Hence, the analysis can safely follow local worst-case paths only. The ARM7 is one example architecture of this class. On a timing accident all components of the pipeline are stalled until the accident is resolved. This even allows for a much simpler analysis where architecture components (e.g., cache, bus occupancy, etc.) can be analyzed separately (i.e., a safe parallel decomposition of the WCET problem is feasible).
- *Compositional Architectures with k-bounded effects:* The architecture suffers from k-bounded timing anomalies but not from domino effects. In general, an analysis has to consider all paths. To trade precision with efficiency, it would be possible to safely discard local non-worst-case paths by adding a constant to the computed WCET bound, as discussed in Section 3.3. So far, no non-fully timing compositional architecture has been formally proven to belong to this class.
- *Non-compositional architectures:* Architectures belonging to this class exhibit domino effects. The MPC755 is known to belong to this class of architectures, because its complex pipeline might cause a domino effect (see Section 3.3). For such architectures timing analyses always have to follow all paths since any local effect may influence the future execution arbitrarily.

4 MRU Domino Effect

This section discusses the most-recently-used replacement strategy in the context of static timing analysis. In contrast to the LRU replacement algorithm, MRU discards the most-recently accessed cache line upon a cache miss. The MRU replacement algorithm is most useful when older data is likely to be reused (e.g., after sequentially scanning an array or a file for data) [3].

Figure 3 demonstrates the domino effect by means of a 2-way cache using the MRU replacement strategy. In this example, the memory locations a and b are accessed in an alternating pattern. Starting with an empty cache, the cache set contents stabilize after two accesses. After the first two accesses

[2] Assuming the constant k is known.

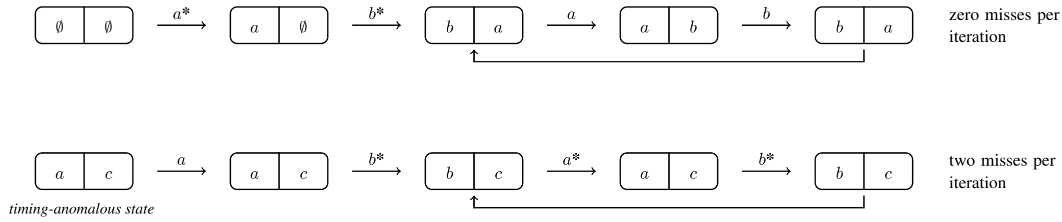

Figure 3 An example domino effect for a 2-way cache using an MRU replacement policy for the repeating access sequence $(a, b)^+$. The left-hand side of a set depicts the most-recently accessed element. The first row features an empty initial cache state, where no misses occur for the given sequence. The second row demonstrates a different initial cache state that causes all accesses except the first to miss the cache. Each miss is marked by *.

that miss the cache set the access sequence will only produce hits. Starting with a cache set that contains the addresses a and c, where a is the most-recently accessed one, each access to the cache except for the first will lead to a cache miss. Because the MRU policy retains older data (i.e., the memory location c in this case), an access to a will evict b from the cache and vice versa.

Proof. **(The MRU algorithm exhibits a domino effect)** Let $n \in \mathbb{N}_{\geq 2}$ be the associativity of a cache governed under the MRU replacement policy. Additionally, let $h, m \in \mathbb{R}_{\geq 0}$ with $h < m$, where h and m are the costs for cache hit and cache miss, respectively.

To show the presence of a domino effect, we need to find a path through a program and two initial hardware states, such that the difference in execution times starting from the two initial states is not constantly bounded.

Let P be a program alternately accessing the distinct memory locations a and b (starting with a) that map to the same cache set and $\vec{\pi}$ be a path through that program. Furthermore, let η_{empty} be a hardware state where the target cache set is initially empty, and η_{full} be an initial hardware state, where the target cache set contains the disjoint memory locations $a, m_1, m_2, ..., m_{n-1}$ with $m_i \neq b$ for $i \in \{1, 2, ..., n-1\}$ and where the cache line containing a is the most-recently accessed one.

It holds:

$$\gamma_{\vec{\pi}}(\eta_{empty}, i) = \begin{cases} m & i \in \{1,2\} \\ h & \text{otherwise} \end{cases}$$

$$\gamma_{\vec{\pi}}(\eta_{full}, i) = \begin{cases} h & i = 1 \\ m & \text{otherwise} \end{cases}$$

Furthermore, it holds $\gamma_{\vec{\pi}}(\eta_{full}, 1) < \gamma_{\vec{\pi}}(\eta_{empty}, 1)$ and $\Gamma_{\vec{\pi}}(\eta_{full}, l) \geq \Gamma_{\vec{\pi}}(\eta_{empty}, l)$ for all $l > 2$. Thus, the state η_{full} is timing-anomalous, in accordance to Definition 5.

The timing anomaly is not k-bounded. For any $k \in \mathbb{R}_{\geq 0}$ we can choose an $l \in \mathbb{N}, l > 2$, such that the k-boundedness according to Definition 6 does not hold:

$$\begin{array}{rccl} & \Gamma_{\vec{\pi}}(\eta_{full}, l) - \Gamma_{\vec{\pi}}(\eta_{empty}, l) & \leq k & \quad | \quad \Gamma_{\vec{\pi}}(\eta_{full}, 3) - \Gamma_{\vec{\pi}}(\eta_{empty}, 3) = 0 \\ \Leftrightarrow & \sum_{i=4}^{l} (\gamma_{\vec{\pi}}(\eta_{full}, i) - \gamma_{\vec{\pi}}(\eta_{empty}, i)) & \leq k & \\ \Leftrightarrow & \sum_{i=4}^{l} (m - h) & \leq k & \\ \Leftrightarrow & (l-3)(m-h) & \leq k & \\ \Leftrightarrow & l & \leq \dfrac{k}{m-h} + 3 & \end{array}$$

For $l > \left\lceil \frac{k}{m-h} \right\rceil + 3$ the above relation is false. ◀

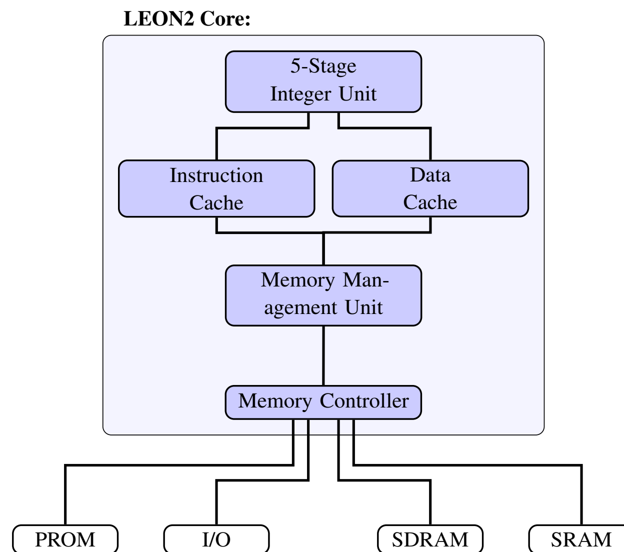

Figure 4 *Simplified block diagram of the LEON2 architecture.*

5 LEON2 Timing Anomaly

In this section we discuss the LEON2 hardware architecture. The LEON2 was developed at Aeroflex Gaisler as a successor of the ERC32 processor. A radiation-hardened version of the LEON2 is available [1] which makes it suitable for the space domain, where fault-tolerance is required.

Similar to the ARM7, the LEON2 features a rather simple pipeline. The pipeline comprises five stages. To speed up execution the LEON2 comprises disjoint instruction and data caches. Figure 4 depicts a block diagram of the LEON2 showing the memory hierarchy.

On a first view, the LEON2 appears to be a fully timing compositional architecture. The processor neither performs speculative fetching nor does it execute instructions speculatively. The LEON2 does not possess any branch-prediction mechanism. Instruction are executed and completed in-order. Each instruction has to visit the five pipeline stages one after another. Thus, an instruction cannot overtake a slower instruction blocking a certain pipeline stage. This prevents the possibility of a scheduling anomaly. Upon a timing accident (i.e., a cache miss) the internal pipeline is stalled until the accident is resolved. Both caches commonly use the LRU replacement policy[3], which is known to behave in a timing compositional manner. It appears that none of the above described timing anomalies can occur. However, in the following we will show that the LEON2 has a hardware feature that potentially triggers a timing anomaly (depending on the system state).

Upon a cache miss, the processor needs to load the missing cache line from main memory. Usually, the whole cache line is loaded and put into the cache. Until the cache line has been filled, the processor stalls the originating memory access. To reduce latencies, some architectures start loading the cache line at the requested address directly forwarding the received data to the core (*cache streaming*).

A similar technique is available in the LEON2 architecture. Each cache line is equipped with valid bits for each word[4] inside the cache line. A cache line is either 16 or 32 byte wide and thus

[3] The LEON2 is synthesized from a VHDL model where different replacements algorithms can be configured. Among others, MRU is a possible choice.

[4] A word is four bytes on the LEON2 hardware architecture.

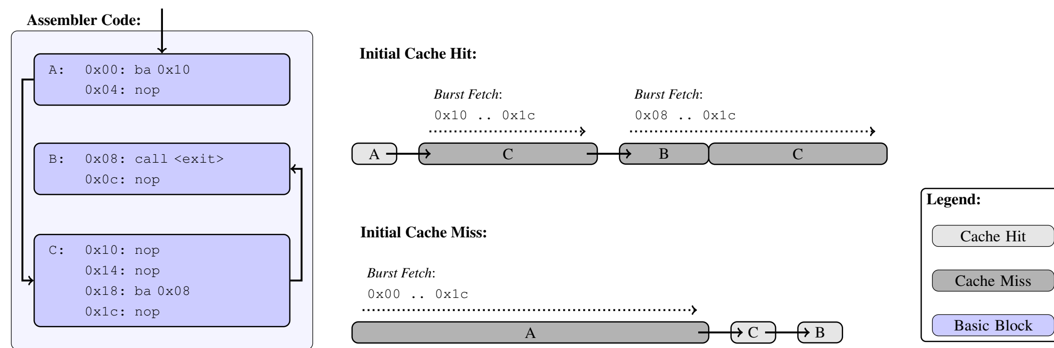

Figure 5 *LEON2 timing anomaly*: The example demonstrates a timing anomaly present in the LEON2 processor caused by the instruction cache line fill mechanism. The basic blocks A, B, and C reside in the same cache line. The local best case — assuming a cache hit for the instructions in basic block A — causes the global worst case execution of the example: The core performs ten instructions fetches. On the contrary, only eight instruction fetches are issued upon an initial cache miss.

comprises either four or eight valid bits. Upon a data cache miss, solely the requested word is loaded from memory and put into the corresponding cache line. The instruction cache operates slightly differently than the data cache. If an instruction fetch misses the code cache, the processor burst-fills the corresponding cache line starting from the requested instruction till the end of the line. The processor does not issue wrap-around burst fetches. Consequently, cache lines might only be filled partially. Furthermore, the processor does not check for existing entries upon burst-filling the cache line. A timing anomaly finally becomes possible, as the LEON2 processor allows cache line fills to be interrupted under certain circumstances.

Figure 5 demonstrates how the described cache line fill mechanism can trigger a timing anomaly. In this example the contents of the cache are assumed to be initially unknown. Each cache line can hold up to eight instructions. Assuming an initial cache miss, the core fills the whole cache line. All in all, the processor issues eight instruction fetches. Assuming cache hits for the first two instruction fetches (basic block A) causes a timing anomaly. The remainder of the target cache line still remains unknown. Reaching the basic block C, a static analysis then would need to assume a cache miss. Recall that the processor might abort a cache line fill operation. Thus, the instructions of basic block C need not necessarily be cached, although cache hits have been assumed for the initial accesses to the cache line. In this case, the core will fill the upper half of the target cache line. Eventually, the program branches to the basic block B. Again, a static analysis would need to assume a cache miss. Because the processor does not check whether burst-fetched instructions are already cached, the instructions in basic block C will be fetched again. Altogether the core performs ten fetches after the initial cache hits. So, the processor performs 20% more memory accesses under the initial assumption.

Despite the simple structure of the LEON2 a timing anomaly is possible, caused by a rather simple, average-case performance increasing hardware feature. Obviously, the timing anomaly is a speculation timing anomaly (see Section 3.5). Fetching subsequent instructions upon an instruction cache miss, the processor assumes a sequential execution of the program.

The described timing anomaly is k-bounded. It is easy to see that the described effect will eventually stabilize – a positive side effect of the employed cache replacement policy. Due to space limitations we can not include the proof in this paper.

The code structure that causes the timing anomaly depicted in Figure 5 is not uncommon. Analyzing industry LEON2 software, we were able to verify that due to the code structure this

phenomenon might occur in real world applications. Due to a non-disclosure agreement we must not provide further details about this particular software.

6 Conclusion

In this paper we have proved that the MRU replacement policy is prone to domino effects. By this, we have completed the list of commonly used replacement policies suffering under timing anomalies. Additionally, we have shown that the LEON2 exhibits a timing anomaly, despite its simple structure.

In the future, we plan to study other hardware architectures that are being used in the automotive and the aerospace domain. Furthermore, we will check whether known instances of timing anomalies can be proven to be k-bounded.

References

1 AEROFLEX GAISLER. `http://www.gaisler.com`.

2 Christoph Berg. PLRU cache domino effects. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Dresden*, number 06902 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), July 2006.

3 Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB 1985: Proceedings of the 11th international conference on Very Large Data Bases*, pages 127–141. VLDB Endowment, 1985.

4 Jochen Eisinger, Ilia Polian, Björn Becker, Stephan Thesing, Reinhard Wilhelm, and Alexander Metzner. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *DDECS '06: Proceedings of the 2006 IEEE Design and Diagnostics of Electronic Circuits and systems*, pages 13–18, Washington, DC, USA, 2006. IEEE Computer Society.

5 Jakob Engblom and Bengt Jonsson. Processor pipelines and their properties for static WCET analysis. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 334–348, London, UK, 2002. Springer-Verlag.

6 Raimund Kirner, Albrecht Kadlec, and Peter Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 119–128, Dublin, Ireland, July 2009. IEEE.

7 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium (RTSS)*, December 1999.

8 Jan Reineke and Rathijit Sen. Sound and efficient wcet analysis in the presence of timing anomalies. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

9 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, July 2006.

10 Jörn Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

11 Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

12 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, July 2009.