Formal Executable Models for Automatic Detection of Timing Anomalies

Mihail Asavoae
CEA LIST
Gif-sur-Yvette, France
mihail.asavoae@cea.fr

Belgacem Ben Hedia
CEA LIST
Gif-sur-Yvette, France
belgacem.ben-hedia@cea.fr

Mathieu Jan
CEA LIST
Gif-sur-Yvette, France
mathieu.jan@cea.fr

Abstract

A timing anomaly is a counterintuitive timing behavior in the sense that a local fast execution slows down an overall global execution. The presence of such behaviors is inconvenient for the WCET analysis which requires, via abstractions, a certain monotony property to compute safe bounds. In this paper we explore how to systematically execute a previously proposed formal definition of timing anomalies. We ground our work on formal designs of architecture models upon which we employ guided model checking techniques. Our goal is towards the automatic detection of timing anomalies in given computer architecture designs.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Computer systems organization → Embedded systems

Keywords and phrases timing anomalies, predictability, formal methods, model checking

Digital Object Identifier 10.4230/OASIcs.WCET.2018.2

Acknowledgements The authors would like to thank Simon Wegener from AbsInt GmbH for providing valuable feedback on this work.

1 Introduction

Modern computer architectures are designed to alleviate the bottleneck between processors, and memory systems, leading to utilization of caches, pipelines and speculation mechanisms. Such architectures are often used in embedded system design and hence required to satisfy, *a posteriori*, stringent timing behavior. The alternative is to design predictable systems, which focus on building systems with *a priori* guarantees of timing requirements.

The quest for predictability is a complicated endeavor as all components to build and execute a system, such as processors, high/low-level languages, compilers, operating systems, communication systems, etc., can impact the definition and verification of timing requirements. Designs of predictable systems as well as associated timing analyses identify and circumvent the sources of non-predictability in various ways: disabling the problematic component(s) (e.g., a particular shared resource), proposing timewise restrictions (e.g., semantics based on temporal isolation), using predictable components (e.g., LRU caches) or even straightforwardly

18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).
Editor: Florian Brandner; Article No. 2; pp. 2:1–2:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

assuming the predictability. Timing analyses need to overcome various challenges: in single-cores, the worst-case execution time (WCET) analysis is complicated by the presence of timing anomalies [11] whereas in multi-cores, the worst-case response time (WCRT) analysis is hampered by timing compositionality issues [5].

The WCET analysis computes sound and (desirably) tight worst-case execution bounds, exploring, via convenient abstractions, all the execution paths of a program running on a computer architecture. Typically, the WCET analysis works on the control-flow graph of the binary code, augmented with semantic information from both the code (e.g., loop bounds) and the underlying architecture (e.g., cache hits/misses). The WCET analysis workflow integrates the results of cache analyses with accurate pipeline modeling to safely search for the longest execution path. This searching process is complicated by the presence of timing anomalies as these are non-monotonic behaviors. In essence, a timing anomaly is a counterintuitive behavior in the sense that the local worst-case timing behavior does not result in the global worst-case performance.

Formal-based methods, e.g., static analysis or model checking, soundly explore all system behaviors, while mitigating precision and performance arguments. Formal reasoning could either evaluate predictability issues of existing systems or guide the construction of predictable systems. Hence the formal, systematic study of timing anomalies becomes essential. In this direction, the first formal definition of timing anomaly is proposed in [14]. *The contribution of this paper is to execute this formal definition of a timing anomaly*, based on model-checking, *towards the automatic detection of timing anomalies*. Our method consists of three phases. First, we consider formal executable models of particular computer architectures, specified using the TLA+ language [8]. These models are deterministic and tested for conformance against actual system behaviors. Second, we systematically enable, directly over the concrete models, non-deterministic choices (i.e., abstract behaviors) as the necessary conditions to facilitate the study of timing anomalies. Finally, we employ model checking, using the TLC tool [18] for TLA+ models, to explore the execution paths of the abstract specification. While our method for automatic detection of timing anomalies is general, our current investigation is in its incipient stages. However, we evaluate our technique on standard examples of scheduling timing anomalies while using models of resource contention in superscalar processors.

We organize this paper as follows. In Section 5, we review some related work and in Section 2, the formal definition of timing anomalies. In Section 3 we briefly introduce the TLA+ language and present our formal architecture models. In Section 4 we describe the automatic detection of timing anomalies. We conclude and outline future work in Section 6.

2 Timing Anomalies – Definition and Examples

Essentially, the first formal definition of timing anomalies, in [14], encodes *an abstract state space*, constructed with respect to both an architecture and an input program and *a property pattern*, expressed with respect to a locality concept. Our proposed method *executes* this formal definition, towards an automatic technique for the detection of timing anomalies. Next, we introduce the ingredients: the running examples and the necessary steps to formalize the timing anomalies.

We consider as running examples the cases of scheduling timing anomalies introduced in [16]. The goal is to study policies of resource allocation (e.g., functional units) in superscalar architectures. Two snapshots of the execution stage of superscalar processors are shown in Figure 1. On the left side, the resources FU_1 and FU_2 execute instructions in the program order, while on the right side, the reservation stations $RS1_{1,2}$ and $RS2_{1,2}$ allow out-of-order

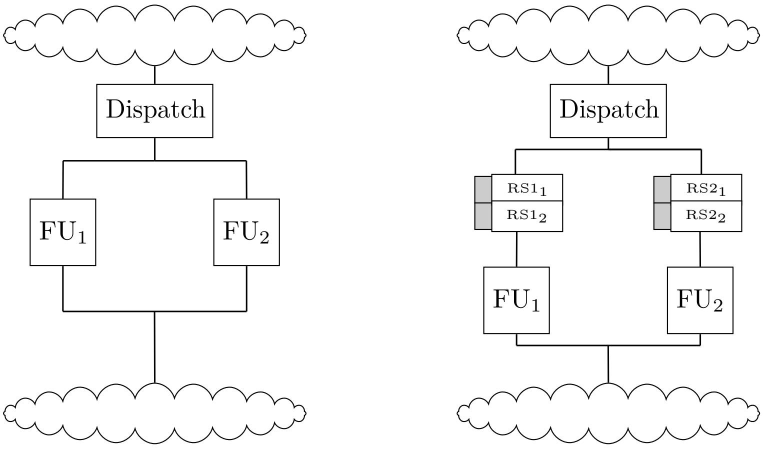

Figure 1 Snapshot of superscalar processor with: (left) in-order resource allocation for both FUs and (right), respectively out-of-order resource allocation for both FUs.

execution on both FU$_1$ and FU$_2$. On these platforms we execute program paths of size 4 (i.e., instructions A to D with the alphabetical order giving the program order), under certain allocation constraints, as in Figure 2.

The architecture model in Figure 1 (left) is without reservation stations and the resource allocation is dynamically decided for instructions like A, based on resource availability. The resource FU$_1$ is the default execution unit for such instructions in the case when both FUs are available. Consequently, the program execution is guided by the program order. The architecture model in Figure 1 (right) imposes different constraints on the set of instructions with respect to resources. As supported by the constraints in Figure 2 (right), each instruction could be executed on a single type of resource. The resources FU$_1$ and FU$_2$ can also execute instructions in out-of-order fashion, based on the content of their respective reservation stations. Consequently, the program execution is guided by the data dependencies between instructions. In our example, the instructions B and C are independent and could be executed in any order, whereas instructions A and B are always executed in the program order.

Examples of scheduling timing anomalies for the architecture models with in-order and out-of-order resource allocation are shown in Figure 2 (left) and respectively (right). In both cases, a pivot instruction with variable latency causes a timing anomaly. For example, a faster execution of instruction B frees FU$_2$ for the execution of instruction C. It further delays the instruction D whose execution is conditioned by the availability of the same resource FU$_2$, in Figure 2 (left).

The formal definition of timing anomalies requires the following concepts:

(1) an *(abstract) architecture model* to provide the settings of the execution environment;
(2) a notion of *locality* to express local worst-case behaviors;
(3) a path mapping as a *labeling function* to correlate the program with the architecture.

Each of the three points requires specific assumptions. For example, the key ingredient towards the construction of a convenient architecture model - point (1) is to enable non-deterministic choices as a standard method to compactly encode system behaviors. The code and the related input data are also part of the system (abstract) state. Point (2) defines the locality as the sequence of abstract system states which satisfies particular constraints with respect to system behaviors. For example, a system execution path is studied locally – between two points of interest – (e.g., when instruction A is in a particular pipeline stage)

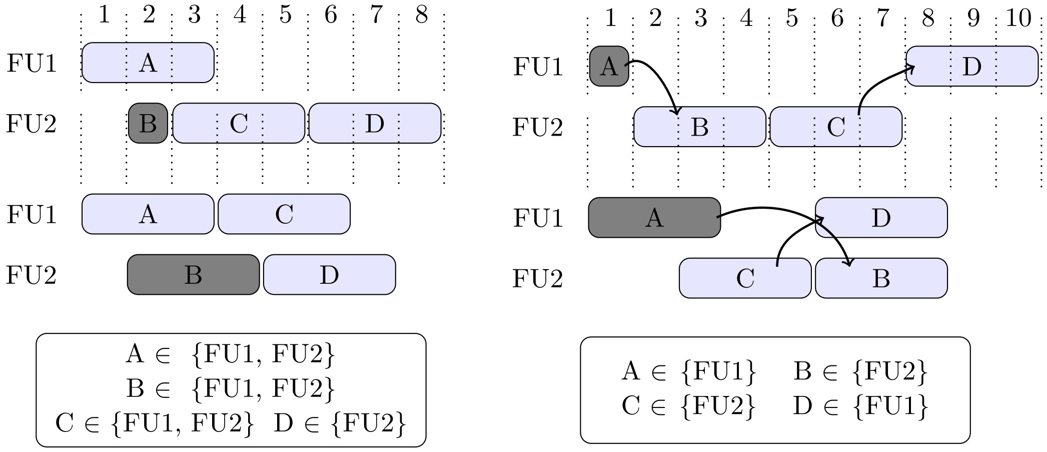

Figure 2 Example of scheduling timing anomalies from [16], with out-of-order (left) and in-order (right) resource allocation, under given allocation constraints for instructions A to D.

with respect to locality constraints (e.g., the interaction between A and all other possible combinations of instructions). Lastly, point (3) is necessary to interpret the search for timing anomalies on the specified architecture system. It relates the instruction-level view given by the program paths with the cycle-level view of the architecture execution paths.

Our approach towards the automatic detection of timing anomalies encodes this formalization. Briefly, we address point (1) when we define, using the TLA+ specification language, a cycle-accurate computer architecture specification; we refer to it as the concrete architecture model. Furthermore, we abstract this concrete model as we encode the necessary non-deterministic choices; we refer to the new specification as the abstract architecture model. Then, we address point (2) when we consider the locality as defined by a particular pipeline stage, hence the locality is a priori encoded by our abstract/concrete architecture state. The locality constraints are either directly represented in the model (as constraints on the input data/program) or computed during the exploration of the state space. Finally, we directly insert the labeling function, i.e., point (3) in the abstract architecture model, more specifically in the code component of the abstract model state. We elaborate next on all these points.

3 Design of Formal Executable Models

Our modeling for automatic detection of timing anomalies fully adheres to the formalization steps (1) – (3), which are required by the definition of timing anomalies from [14]. Our concrete and abstract models are TLA+ specifications.

We choose the TLA+ modeling language because of several semantic considerations. TLA+ features an advanced module system based on interfaces, parameters, local declarations etc. which allow accurate construction of (concrete and abstract) architecture models from simpler components. The modeling language also features untyped set theory (and predicate logic) to specify rich state information. Abstraction in TLA+ is ensured by temporal existential quantification which hides unnecessary state elements. Refinement in TLA+ is ensured by supporting stuttering invariance (i.e., execution steps that do not change the values of state variables of interest) which allows reasoning about system paths on different levels of granularity. All the aforementioned concepts establish TLA+ as an unified logical language designed to specify both systems and their properties, as well as verifying, using the same specification, both a system and its possible refinements. This latter characteristics of TLA+ is particularly attractive for our investigation towards automatic detection of timing anomalies as our framework is based on a single formal specification (i.e., of the concrete hardware model), which is then systematically refined. We recommend [12] for an in-depth and comprehensive survey of the TLA+ language semantics and its applications. Next, we introduce several elements of the TLA+ language and we exemplify their usage with snapshots of our formal models.

A TLA+ specification is two-tiered. The first level contains state and state transition formulas (i.e., system specification) and the second level contains temporal formulas evaluated on sequences of states (i.e., system properties). A particularity of the TLA+ language is the transition predicate (also called *action*) which establishes a relation between variable values in the current and next states. For example, if x is a state variable, the action $x' = x + 1$ means that the next value of x (the primed variant) is the current value of x (the unprimed variant) incremented by 1. Whenever state elements are unmodified by a transition, for example $x' = x$, the TLA+ notation is *UNCHANGED x*. If x is a record with two fields *fst* and *snd*, an individual field is accessed with *"."*, for example *x.fst*. As such, the TLA+ action $x' = [\ x\ EXCEPT!.fst = 1\ ]$ means that only the value of *fst* of x is modified in the next state. When a TLA+ module X with an internal state variable x and a transition *Act* is used in another module, the operator *"!"* gives access to each, e.g., *X!x* and respectively *X!Act*. Finally, we denote by $\langle\ S\ \rangle$ the state configuration of an TLA+ specification *Spec* (i.e., S is the set of semantic entities that are necessary to define the behaviors of *Spec*).

(1) The hardware model – concrete

We define the two instances of superscalar architectures from [16] and for each instance we define a concrete model which is then systematically transformed into an abstract model. The formal computer architecture model is developed in a modular fashion, according to the principles described in [9], using the TLA+ module system.

The state configuration $\mathcal{C}$ of our concrete architecture model consists of two state components: the architecture *Arch* and the input program *Code*.

$$\mathcal{C}\ =\ \langle\ Arch,\ Code\ \rangle\ .$$

The concrete *Arch* consists of several variables to represent the pipeline stages; these variables are updated cycle-wise based on the content of their inner states and the necessary signals from the memory system, as in [16]. Since we aim for the detection of scheduling timing anomalies, we implicitly represent the signals from the memory system, while fully specifying the execution pipeline stage and an instruction progress through the pipeline. The *Arch* state configuration for the architecture model in Figure 1 (left) is that of a standard 5-stage pipeline:

$$Arch\ =\ \langle\ \_IF,\ \_ID,\ \_EX,\ \_MEM,\ \_WB\ \rangle\ .$$

whereas for the architecture model in Figure 1 (right) is a 6-stage pipeline, with an extra instruction issue stage. The names for the pipeline stages stand for instruction fetch (*_IF*), instruction decode (*_ID*), execute (*_EX*), memory access (*_MEM*) and write-back (*_WB*).

Both pipeline models are dual-issue. Structurally, our architecture models are incrementally built from simple parameterized modules of buffers and functional units, which are instantiated into pipeline stages. Each functional unit and internal buffers of the pipeline

$$
\begin{array}{ll}
 & \textbf{AcquireFU1} \triangleq \\
 & \quad \land\ condAcquireFU1 \\
(1) & \boxed{\begin{array}{l} \land \text{ IF } isCurrIns \\ \quad \text{THEN } \_IF' = [\ \_IF \text{ EXCEPT } !.buff = FBUFF!Set(code.currIns)] \\ \quad \text{ELSE } \ \_IF' = [\ \_IF \text{ EXCEPT } !.buff = FBUFF!Reset] \end{array}} \\
 & \quad \begin{array}{ll} \land\ \_ID' & = updateID\ (\_ID) \\ \land\ \_EX' & = [\ \_EX \text{ EXCEPT } !.fu1 = FU1!Acquire(\_ID.buff.instr)\ ] \\ \land\ \_MEM' & = updateMEM\ (\_MEM) \\ \land\ \_WB' & = updateWB\ (\_WB) \\ \land\ \_code' & = updateCode\ (code) \\ \land\ cycle' & = updateClk\ (cycle) \end{array} \\
 & \\
(2) & \boxed{\begin{array}{l} \land \text{ IF } isCurrIns \\ \quad \text{THEN } \exists\ \mathbf{d} \in \mathbf{code.currInstr.tvar:} \\ \qquad\qquad \_IF' = [\ \_IF \text{ EXCEPT } !.buff = FBUFF!Set(code.currIns,\ \mathbf{d})] \\ \quad \text{ELSE } \ \_IF' = [\ \_IF \text{ EXCEPT } !.buff = FBUFF!Reset] \end{array}}
\end{array}
$$

Figure 3 The TLA+ rule for acquiring the functional unit FU1. With (1), the rule presents the concrete architecture behavior. When (1) is replaced by (2), the rule shows the abstract architecture behavior when exploiting timing variations for the current instruction.

stages provide an interface for theirs operations, accessible via the "!" operators. Semantically, our architecture model for the out-of-order resource allocation supports the Tomasulo algorithm, as in [1], whereas the in-order resource allocation is driven by the program order.

Let us briefly explain our concrete architecture model using an excerpt of the TLA+ formal model, in Figure 3. We recall that our objective is to study scheduling timing anomalies which manifest when instructions are deployed for functional units in the *execute* stage of the pipeline. This scheduling mechanism consists of operations of acquire and/or release of one or both functional units (i.e., in short *FUs*). Figure 3 presents the specification of acquiring the functional unit FU1, a rule named **AcquireFU1**. Other TLA+ rules specify pipeline stalls, flushes, simultaneous acquires of both FUs, etc. Each rule is guarded by a predicate (e.g., *condAcquireFU1*) and contains the actions to update the *Arch* and *Code* (i.e., variable *_code*) parts of the concrete configuration, as well as the clock variable (i.e., *cycle*). In our example, the guard *condAcquireFU1* is a predicate which establishes the necessary conditions to activate the rule **AcquireFU1**:

$$
\begin{array}{ll}
\textbf{condAcquireFU1} \triangleq & \land\ \neg\ emptyID\ \ \land\ \ isAvaiFU(\_ID.buff.instr,\ FU1!fname) \\
 & \land\ \ (emptyEX\ \ \lor\ \ (emptyFU1\ \ \land\ \ FU2!inExec(\_EX.fu2)))
\end{array}
$$

The first line ensures that there is an instruction in the decode stage which is ready and needs to be executed by FU1 as *isAvaiFU* checks whether instruction *instr* from the decode stage can be executed over the functional unit FU1. The second line ensures that there is not another case of acquire or release of either FUs at the same time.

When *condAcquireFU1* is true, the new content of the instruction stage (emphasized by (1)), *_IF'*, retrieves a new instruction, if it exists (variable *isCurrIns*), and sets the internal state of this stage (using the accessor *".buff"*) to this instruction. If a new instruction is not available, the new internal state of the instruction stage is reset, i.e. it is emptied, using the operation *Reset*. The new content of the execute stage, *_EX'*, is modified only for the first functional unit (using the accessor *".fu1"*) with an instruction from the decode stage (i.e., *_ID.buff.instr*). In a similar way, the other pipeline stages (decode, memory access and write-back) and the code update their next state, via the corresponding *update* functions. Finally, the clock cycle advances using the *updateClk* function.

Let us recall that the input program is represented in the concrete state configuration $\mathcal{C}$ by the state component *Code*. In our TLA+ models, the program is represented by its set of program paths and each path is a sequence of instructions. An instruction is represented by several parameters: the program counter, the execution resources (as the set of necessary FUs), the latencies (the concrete representation considers exactly one latency per instruction), the dependencies with respect to other instructions and finally the temporal availability (in this latter case, it is borrowed from the task-oriented model of computation). Whereas our instruction representation does not model a particular instruction set architecture (ISA), it contains all the necessary semantic ingredients to capture existing ISAs semantics. Figure 2 shows examples of instructions respecting the properties of our instruction model.

The concrete architecture models are cycle-accurate and deterministic. The part *Code* of $\mathcal{C}$ is instantiated with concrete program paths and executed using the TLC model checker. We rely on the TLC statistics on the state space to assess the determinism aspect of our architecture models and to drive, whenever necessary, model refinements. We extensively test both concrete models to gain confidence in their correct functionality and determinism. The abstract models are constructed directly over the concrete models, e.g., replacing predicate (1) with (2) in Figure 3. We detail this procedure in the next section.

(2) The locality concept

It is accepted [11, 14] that locality matches an instruction progress through the pipeline stages. The notion of locality is thus formalized as a path fragment of interest, for any execution path in the model. The particular example of scheduling timing anomalies, which appear in processors due to contention for functional units defines the locality level as the execution pipeline stage, i.e. the $\_EX$ stage in our pipeline models.

The locality constraints are convex predicates which hold locally – on path fragments of interest. They could be (a) pre-determined and encoded in the program part of the state configuration, e.g., in our case in *Code*, or (b) dynamically calculated during the model execution. We experiment with both variants and henceforward and without the loss of generality, our locality constraints are given, i.e., we assume (a). Precisely, we work with convex predicates in the form of single linear inequalities where an instruction latency is bounded by a pre-computed value. For example, in Figure 2 (left), the execution time for B is bounded by 1 for the first execution and by 3 for the second execution.

(3) The labeling function

We use the *Arch* configuration to construct cycle-accurate architecture models. It is necessary to relate them to the instruction-level information presented in the *Code* configuration. We address this aspect directly in the concrete architecture model, as our method is centered around the program path. Hence, *Code* encodes all the program paths [10] which are evaluated path by path. In the general form, our code-related configuration is:

$$Code \ = \ \langle \ [Paths], \ CurrPath \ \rangle.$$

with the input program and data in $[Paths]$ and the current program path in $CurrPath$. Since the study of timing anomalies require input variations at the path level, we assume, without loss of generality, that a simplified *Code* contains only $CurrPath$. Structurally, a program path is encoded as a list of instructions. Semantically, each instruction advances in the program order, given by the program counter, through the pipeline stages until *Execute*, where a corresponding resource allocation takes place. TLA+ facilitates a flexible encoding of path-related variations with its set-theoretic semantics. For example, the latency 3 of instruction B from Figure 2 is adequately represented in *CurrPath* by a singleton.

The design of concrete architecture models follows the principles of the formal definition of timing anomalies: the architecture is deterministic and cycle accurate, the code is part of the model; the program paths are evaluated one by one etc. Over such infrastructures, we systematically construct abstract models which are checked for timing anomalies. We present next how we perform abstractions over the concrete model and how we use model checking for the detection of timing anomalies.

4 Detection of Timing Anomalies

Generally, a TLA+ specification *Spec* consists of the definition of the initial state *Init* and a state transformer *Trans* applied over the state variables, e.g., in our case $\mathcal{C}$:

$$Spec \ \ == \ \ Init \ \wedge \Box \ Trans_{\mathcal{C}}.$$

where $\Box$ is the temporal operator “always”. *Trans* contains guarded transitions for pipeline stalls, flushes, single acquire of FU_1 or FU_2, simultaneous acquires of both FUs, simultaneous acquire of FU_1 and release of FU_2 etc.

(1) The hardware model – abstract

We construct an abstract architecture model which augments the concrete model *Spec* with non-deterministic choices. The abstraction creates “diamonds” in the specification which are to be explored with the model checker. The abstract state configuration, $\mathcal{A}$ refines $\mathcal{C}$ in both architecture *AArch* and program *ACode* components:

$$\mathcal{A} \ = \ \langle \ AArch, \ \ ACode \ \rangle \ .$$

For example, variable latency $\{1, 3\}$ of instruction B in Figure 2 (left) form a diamond in the abstract architecture model. Similar variations lead to have *CurrPath* of the *ACode* configuration encoding sets of concrete paths. The state transformer $Trans'$ associated to *AArch* extends its concrete counterpart based on *Arch* to fully explore these sets of paths. For example, for the aforementioned instruction B, the latency is non-deterministically chosen between 1 and 3, when applicable (i.e., in certain states of interest). The model checking explores both possibilities of the new abstract architecture model – $Spec'$:

$$Spec' \ \ == \ \ Init \ \wedge \Box \ Trans'_{\mathcal{A}}.$$

The abstract architecture model includes the path-level variations, as presented in Figure 3 on rule **AcquireFU1** where predicate (2) replaces (1). This particular rule shows two important aspects of our abstract model: it is constructed directly over the concrete model and the abstraction points – the “diamonds” – are visible in the specification. This latter point opens the possibility of exploring the diamonds in a guided way, which establishes the third step of our systematic framework for automatic detection of timing anomalies.

A timing anomaly is characterized by a pair of execution paths because it “compares” local worst-case variations with respect to global worst-cases. For example, let us consider two execution paths, as in Figure 4 (left), where local variations Δ_1 and respectively Δ_2,

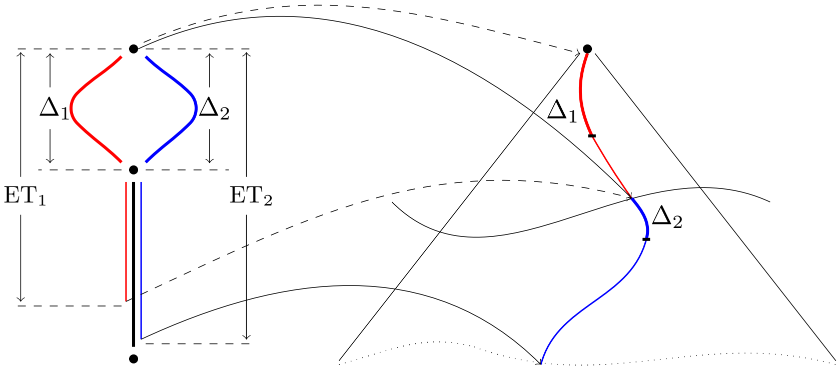

Figure 4 Abstraction diamond (left) and timing anomaly on the search tree (right).

with $\Delta_1 > \Delta_2$ result in global execution times ET_1 and respectively ET_2, with $\mathrm{ET}_1 < \mathrm{ET}_2$. Intuitively, automated detection of timing anomalies over the abstract model *Spec′* means searching for such pairs of execution paths with counter-intuitive behavior. Now, it remains to encode this property in TLA+ and launch the TLC model checker to search for timing anomalies. A simple way to encode this property is as an invariant of the form:

$$Prop_{TA} = \Box \neg (\Delta_1 > \Delta_2 \ \wedge ET_1 < ET_2)$$

and the property is checked on *Spec′*.

We accommodate such a formulation over a transformed search tree, as in Figure 4 (right), where, intuitively, each path consists of two different paths of the original encoding of the search space. More simply, a diamond is fully unfolded along a single path, while respecting the initial conditions of its cases. For example, the two execution paths in Figure 2 (left) form a single execution path in the new search tree, with $\Delta_1 = 1$, $\Delta_2 = 3$, $ET_1 = 8$ and $ET_2 = 7$. This new search tree is constructed on-the-fly and the detection procedure stops when the first "long" path which violates the property $Prop_{TA}$ is found. Intuitively, the diamond unfolding corresponds to a simple observer automaton which toggles between two states (e.g., with a set/reset-like semantics).

We guide the model checker to find "long" paths, implementing a mechanism to track the exploration of all diamonds. We opt to encode this mechanism directly in the abstract model (it can also be automatically generated for a given abstract model). As such, we extend the abstract configuration $\mathcal{A}$ to accommodate the guiding mechanism:

$$\mathcal{A}_{state} \;=\; \langle\ AArch,\ ACode,\ Guide\ \rangle\ .$$

In its simplest form, *Guide* monitors the execution, records taken decisions and direct subsequent executions to the unexplored state space. Precisely, our *Guide* encodes how to construct long paths and then how to fully explore the new state space. We address the first point using a single TLA+ rule which is activated only when the first (red) execution in Figure 4 (right) terminates and under the same initial conditions, the second (blue) execution starts. With respect to the second point, our current implementation supports a rudimentary, though systematic, exploration of all diamonds in our abstract model. For example, if the set of timing variations of a particular instruction is $\{2, 4, 5\}$, *Guide* explores (in this order) the diamonds $\{2, 4\}$, $\{2, 5\}$ and $\{4, 5\}$. Variations on multiple instructions are handled in a similar fashion with the order of diamonds also depending on the instruction program counters. More refined heuristics to speed-up the model checking (e.g., with interpolation techniques as in [6]) are left for future work.

We detect (scheduling) timing anomalies like those in Figure 2, from [16]. We experiment with small scale architecture models, in total around 2K lines of TLA+ specification, upon which we execute both concrete and abstract program paths. At the architecture level, we consider two dual-issue pipelines with 5 stages for the in-order functional unit allocation and respectively with 6 stages for the out-of-order variant, with precise modeling of the instruction advancement in the pipelines and with complete specification of resource contention in the execution stages. We also perform preliminary experimentation with variants of the in-order architecture models based on pipeline stalls, as indicated in [4]. At the program level, we consider program paths which activate the worst-case contention scenarios for the model in Figure 1 (right), when all reservation stations and functional units are full. Next, we elaborate on some experimentation, conducted on a quad-core Intel i7 at 2.8GHz with 16GB RAM and with the TLA+ Toolbox using the TLC model checker version 2.19.

Let us exemplify with the following test scenario, named $\mathcal{T}$, upon which we construct several test variants. At the architecture level, we consider the 5-stage pipeline with in-order functional unit allocation. At the code level, we use a program path of size 20, with multiple variations for instruction latencies and resource allocations (actual statistics on the size of the both feasible and infeasible search space are subsequently given). We investigated several aspects of our approach: (a) the concrete executions are deterministic, (b) the absence of timing anomalies in $\mathcal{T}$ and finally (c) the detection of timing anomalies in methodically-constructed variants of $\mathcal{T}$, using the guide mechanism. The TLC model checker provides several statistics on the search space, notably the problem diameter, the number of existing states and the number of distinct states. Our extensive evaluation of (a), on concrete executions (i.e., the instruction latencies are given as singletons) of $\mathcal{T}$ end, after 2-3 seconds, with identical numbers on all these parameters. The absence of timing anomalies (b) requires full exploration of the state space of $\mathcal{T}$. As such, we employ bounded model checking (with a bound value of 100) and prove that $\mathcal{T}$ does not have timing anomalies in approximately 7 hours and with a maximum memory consumption of 39GB. The statistics on the entire state space of $\mathcal{T}$ include 839M states found with 835M distinct states (i.e., around 0.5% duplicated states). Finally, we address (c) the detection of timing anomalies in $\mathcal{T}$, using *Guide*. We produce several variants of $\mathcal{T}$, “inserting” timing anomalies (as variations of instruction latencies) into the test scenario. For example, small timing variations (i.e., $|\Delta_1 - \Delta_2| \leq 2$ cycles), at various path locations (i.e., program counters of 5, 14 and 20) cause timing anomalies with *ET*s variations of up to 20 cycles. The timing anomalies are found as ”long” paths in the search tree of Figure 4 (right) using bounded model checking with the bound value of 1000. The running time varies between 1-2 minutes, with around 10M states covered. We also experimented with variants of $\mathcal{T}$ with well-concealed timing anomalies, yielding a running time time of around 1 hour and up to 200M explored states.

We address next some advantages and weaknesses of our detection algorithm. We present a general method, which it is not restricted to scheduling timing anomalies, as exemplified here. Because our approach is constructed over a concrete architecture model, it is possible to subject the detection of timing anomalies to guided, but non-exhaustive, heuristics (as they were firstly observed in [11]). Also, our detection procedure could be tuned to compute the local variations (deltas) of [13]. Finally, our formal architecture models could be adapted to experiment with newly proposed and/or predictability-driven architecture modifications [4]. On the other hand, our approach relies on two daunting tasks: the construction of the formal infrastructure and the handling of the state space explosion. While we discussed possible approaches towards the latter, building a cycle accurate formal executable architecture remains complicated. A possible solution is to use already constructed formal models, another is to automatically extract them from existing HDL designs (as suggested in [1, 3]).

5 Related Works

The first assessment of timing anomalies in the context of the WCET analysis is presented in [11]. It reports timing anomalies caused by caches (and identified in [14] as speculation timing anomalies) in out-of-order architectures. The first formal definition of timing anomalies is proposed in [14]. We elaborate on its technical aspects in Section 2 as it establishes the foundation of our work. A class of timing anomalies (and identified in [14] as scheduling timing anomalies) is studied in [16] on two superscalar models with in-order and respectively out-of-order resource allocation. The work in [2] presents an actual computer architecture – the LEON2 processor – with timing anomalies (i.e., speculation timing anomalies).

Our approach shares similarities with two other approaches [1, 3] on the formal investigation of timing anomalies. Briefly, the work in [1] focuses on proving the absence of timing anomalies using bounded model checking, whereas the work in [3] combines static analysis with measurement-based techniques towards detection of timing anomalies. Both approaches, as well as ours, follow a similar pattern – the construction of a convenient representation of the abstract architecture state space and work, as in our case, under the assumption that the input program has a finite number of paths. With respect to [1], our framework targets the detection of timing anomalies, hence our *Guide* (through simple) advances on the straightforward model-checking algorithm of [1]. Moreover, our framework checks the presence of timing anomalies on a single model and with the property $Prop_{TA}$ given as an invariant, whereas the technique in [1] requires two models out of which one is assumed without timing anomalies and with a property expressed over the execution paths of both models. Also, the work in [1] focuses on identification of timing anomalies independently from a given program, but no complexity analysis or runtime performance results are reported and no specifics of the formal models are presented. We instead focus on the identification of code-specific scheduling timing anomalies and provide details on the formal models in Sections 3 and 4. Note that we could also add constraints in our work to upgrade to a code-independent problem. However, we believe that the code-specific problem is more interesting from an industrial point of view as most hardware architectures are subject to timing anomalies. Identifying where within a code such anomalous behavior can happen are useful to later insert mitigation mechanisms. With respect to [3] which checks timing variations of the worst-case path (computed with an WCET analyzer) using program runs on the actual architecture, our approach directly integrates the concrete architecture model in order to support such runs. The approach in [3] constructs a prediction graph which is a compact representation of instruction-level simulations of program paths. Whereas the work in [3], though extensive, relies on non-exhaustive investigation of the architecture, ours is able to provide formal guarantees with respect to it (though subjected to scalability issues).

The work [11] which introduces timing anomalies in context of the WCET analysis also proposes a simple code modification to eliminate their effects. An alternative approach, in [13] explores the abstract hardware state space using pre-computed local worst-cases called deltas. [7] proposes to speed-up WCET analyses by parallelizing their computations. However, this computation methodology generates timing anomalies that are not necessarily present in the underlying architecture model. Lastly, the compositional timing analyses for multicores, from [4], address the problem of timing anomalies with sound techniques, ranging from pipeline stalls to overapproximation of local effects with integer linear programming.

Automatic detection of timing anomalies supports the design of predictable/compositional systems as, according to [15], the timing anomalies are “at the heart of unpredictability at processor level”. The timing anomalies could have bounded or unbounded effects (also known as domino effects), leading to a classification of computer architectures [17] into: fully timing compositional (i.e., without timing anomalies), compositional with constant bounded effects (i.e., only with bounded timing anomalies) and non-compositional (i.e., with domino effects).

6 Conclusions and Future Work

We presented a methodology to automatically detect timing anomalies based on formal models of computer architectures. Our proposal is systematic; it starts with a concrete architecture model, thoroughly tested to gain confidence in its concrete semantics. Then, we constructed abstractions, which are necessary to facilitate the study of timing anomalies, directly over the concrete architecture model. Finally, we described a detection procedure based on guided model checking. Our preliminary investigation considered a simple transformation of the search space to check for timing anomalies expressed as invariants.

New designs of either whole systems or specific components claim to be free of timing anomalies and it is important to rely on formal techniques to validate their behavior. Our preliminary study remains to be developed in several directions. We leave as our future work a similar investigation of timing anomalies due to prefetching, towards our goal for complete architecture models and the development of heuristic techniques to accelerate the model checking phase (e.g., using cuts, as in [6]).

References

1 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 *DDECS*, pages 15–20, 2006.

2 G. Gebhard. Timing anomalies reloaded. In *WCET*, pages 1–10, 2010.

3 G. Gebhard. *Static timing analysis tool validation in the presence of timing anomalies*. PhD thesis, Saarland University, 2013.

4 S. Hahn, M. Jacobs, and J. Reineke. Enabling compositionality for multicore timing analysis. In *RTNS*, pages 299–308, 2016.

5 S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *SIGBED Rev.*, 12(1):28–36, 2015.

6 J. Henry, M. Asavoae, D. Monniaux, and C. Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *LCTES*, pages 43–52, 2014.

7 R. Kirner, A. Kadlec, and P. Puschner. Precise worst-case execution time analysis for processors with timing anomalies. In *ECRTS*, pages 119–128, 2009.

8 L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

9 M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *SAS*, pages 294–309, 2002.

10 J. Larus. Whole program paths. In *PLDI*, pages 259–269, 1999.

11 T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, pages 12–21, 1999.

12 S. Merz. On the logic of TLA+. *Comp. and Artificial Intelligence*, 22(3-4):351–379, 2003.

13 J. Reineke and R. Sen. Sound and efficient WCET analysis in the presence of timing anomalies. In *WCET*, 2009.

14 J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *WCET*, 2006.

15 L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

16 I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *QSIC*, pages 295–306, 2005.

17 R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.

18 Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *CHARME*, pages 54–66, 1999.