Definitions and Detection Procedures of Timing Anomalies for the Formal Verification of Predictability in Real-Time Systems

Benjamin Binder

université PARIS-SACLAY

Definitions and Detection Procedures of Timing Anomalies for the Formal Verification of Predictability in Real-Time Systems

Définitions et procédures de détection des anomalies temporelles pour la vérification formelle de la prédictibilité des systèmes temps-réel

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580 : Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique
Référent : Faculté des sciences d'Orsay

Thèse préparée à **l'institut LIST (université Paris-Saclay, CEA)**, sous la direction de **Mathieu JAN**, directeur de recherche, LIST (université Paris-Saclay, CEA), le co-encadrement de **Mihail ASAVOAE**, ingénieur-chercheur, LIST (université Paris-Saclay, CEA), le co-encadrement de **Belgacem BEN HEDIA**, ingénieur-chercheur, LIST (université Paris-Saclay, CEA), et le co-encadrement de **Florian BRANDNER**, enseignant-chercheur, LTCI (institut polytechnique de Paris, Télécom Paris)

Thèse soutenue à Paris-Saclay, le 13 décembre 2022, par

Benjamin BINDER

Composition du jury

Membres du jury avec voix délibérative

Alain FINKEL Professeur des universités, ENS Paris-Saclay (université Paris-Saclay)	Président
Stephan MERZ Directeur de recherche, INRIA & LORIA (université de Lorraine)	Rapporteur & Examinateur
Christine ROCHANGE Professeure des universités, IRIT (université Toulouse III - Paul Sabatier)	Rapporteure & Examinatrice
Claire PAGETTI Ingénieure de recherche, ONERA (ISAE-SUPAERO)	Examinatrice

THÈSE DE DOCTORAT
NNT : 2022UPASG086

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title: Definitions and Detection Procedures of Timing Anomalies for the Formal Verification of Predictability in Real-Time Systems

Keywords: Timing Anomalies, Real-Time Systems, Formal Verification, Model Checking, Out-of-Order Pipeline, TriCore

Abstract: The timing behavior of real-time systems is often validated through *timing analyses*, which are yet jeopardized by *timing anomalies* (TAs). A *counter-intuitive* TA manifests when a local speedup eventually leads to a global slowdown, and an *amplification* TA, when a local slowdown leads to an even larger global slowdown.
While counter-intuitive TAs threaten the soundness/scalability of timing analyses, tools to systematically detect them do not exist. We set up a unified formal framework for systematically assessing the definitions of TAs, concluding the lack of a practical definition, mainly due to the absence of relations between local and global timing effects. We address these relations through the *causality*, which we further use to revise the formalization of these TAs. We also propose a specialized instance of the notions for out-of-order pipelines. We evaluate our subsequent detection procedure on illustrative examples and standard benchmarks, showing that it allows accurately capturing TAs.
The complexity of the systems demands that their timing analyses be able to cope with the large resulting state space. A solution is to perform *compositional analyses*, specifically threatened by amplification TAs. We advance their study by showing how a specialized abstraction can be adapted for an industrial processor, by modeling the timing-relevant features of such a hardware with appropriate reductions. We also illustrate from this class of TAs how verification strategies can be used towards the obtainment of TA patterns.

Titre : Définitions et procédures de détection des anomalies temporelles pour la vérification formelle de la prédictibilité des systèmes temps-réel

Mots-clés : Anomalies temporelles, systèmes temps-réel, vérification formelle, vérification de modèles, pipeline out-of-order, TriCore

Résumé : Les systèmes temps-réel sont souvent validés par des analyses temporelles, qui sont mises en péril par des *anomalies temporelles* (AT). Une AT *contre-intuitive* a lieu quand une accélération locale conduit à un ralentissement global, et une AT d'*amplification*, quand un ralentissement local entraîne un ralentissement encore plus grand.
Alors que les AT contre-intuitives menacent le bien-fondé ou la flexibilité des analyses, il n'existe pas d'outils pour les détecter de manière systématique. Nous proposons une structure formelle unifiée pour évaluer les définitions des AT, concluant au manque d'une définition pratique, principalement dû à l'absence de relations entre les effets temporels locaux et globaux. Nous y répondons par la *causalité*, que nous utilisons pour revoir la formalisation de ces AT. Nous proposons aussi une instance des notions spécialisée pour les pipelines out-of-order. Nous évaluons notre procédure de détection subséquente sur des exemples illustratifs et bancs de tests, montrant qu'elle permet de capturer précisément les AT.
La complexité des systèmes exige que leurs analyses gèrent l'important espace d'états résultant. Une solution est de réaliser des analyses *compositionnelles*, précisément menacées par les AT d'amplification. Nous faisons progresser leur étude en montrant comment une abstraction spécialisée peut être adaptée pour un processeur industriel, en modélisant les caractéristiques temporelles clés avec des réductions appropriées. Nous illustrons aussi à partir de cette classe d'AT comment des stratégies de vérification peuvent être utilisées en vue de l'obtention de motifs d'AT.

À mon grand-père, Simon CORMAND (1930-2011)
À ma grand-mère, Colette BINDER (1926-2022)

REMERCIEMENTS

On entend souvent que le travail d'une thèse est synonyme de phases certaines d'isolement voire de perdition. Si j'ai bien sûr dû affronter des difficultés et prendre des décisions pour aller de l'avant, je ne fus jamais seul, pas davantage dans l'adversité que la félicité dont la soutenance est évidemment l'apogée.

Je remercie donc en premier lieu Mathieu, mon directeur de thèse, qui a su conserver son niveau d'exigence tout en me soutenant immanquablement pendant trois ans. Certains sollicitent à l'occation l'avis de leur directeur de thèse, au point parfois d'être capables de mentionner leurs rencontres ; j'ai eu la chance d'avoir un premier encadrant, et toujours disponible.

Ma thèse n'aurait pas été la même sans Mihail, mon encadrant, tant pour ses apports scientifiques et techniques que pour la présence réconfortante d'un ami : je savais que je pouvais compter sur lui de jour comme de nuit. Il m'a permis de relativiser les déconvenues et a offert au déroulement de ma thèse un innocent brin de légèreté.

La rigueur de Florian, mon encadrant, a beaucoup profité à la consolidation de mes travaux. Nous avons passé beaucoup de temps à travailler ensemble au tableau, dès le début de ma thèse : j'ai bénéficié d'une véritable émulation. Je fus maintes fois ravi de pouvoir partager avec lui mes réflexions et mes interrogations sur mes résultats jusque dans leurs moindres détails.

Je remercie également Belgacem, mon encadrant, qui m'a accordé sa confiance dès notre rencontre et qui me l'a renouvelée sans cesse. Il m'a accompagné dans toutes mes démarches, depuis mon parcours d'arrivée au CEA jusqu'à la préparation de ma soutenance, et m'a constamment soutenu et encouragé.

Je pense aussi à tous mes collègues et amis, qui ont grandement contribué à rendre agréables au quotidien ces trois années passées au CEA. Jeux de cartes le midi, nombreuses discussions – scientifiques ou « ordinaires » –, pauses-café, déjeuners : autant de moments partagés pour lesquels je les remercie.

Je tiens aussi à remercier toutes les personnes au CEA qui m'ont régulièrement aidé et permis de mener à bien mes travaux. Je pense en particulier à mes chefs de laboratoire et aux secrétaires du DSCIN. Je remercie Christian GAMRAT, alors responsable scientifique du département, pour son accueil extrêmement chaleureux.

Je remercie ici Nermine de son soutien et ses encouragements indéfectibles et de m'avoir, pour ainsi dire, ouvert la voie. Grâce à elle, j'ai presque vécu deux thèses.

Les derniers mois de rédaction puis finalisation de ma thèse se sont doublés de ma charge d'enseignement en CPGE. Je remercie vivement tous mes collègues de prépa qui m'ont fourni des ressources, m'ont apporté leur aide ou qui ont fait en sorte de ménager mon emploi du temps.

Je remercie M. Alain FINKEL de m'avoir fait l'honneur de présider mon jury, mes rapporteurs Mme Christine ROCHANGE et M. Stephan MERZ d'avoir relu attentivement mon manuscrit, et Mme Claire PAGETTI d'avoir accepté d'examiner mes travaux.

Enfin, je suis reconnaissant à mes parents, mon frère et ma grand-mère de leur aide perpétuelle et ô combien essentielle, jusque dans les aspects logistiques. Je leur dois d'avoir pu focaliser mon attention sur mes travaux et ma thèse.

RÉSUMÉ ÉTENDU EN FRANÇAIS

La problématique de la correction est importante lors du développement des systèmes temps-réel. Outre la correction fonctionnelle, le comportement *temporel* des systèmes temps-réel est souvent vérifié formellement, de façon à garantir que des résultats corrects sont délivrés dans les temps, quelles que soient les conditions d'exécution. Le comportement temporel résulte de la combinaison d'une composante matérielle, dans notre cas un processeur comportant un pipeline d'exécution, et d'une composante logicielle, dans notre cas un programme sous la forme d'une séquence d'instructions. Du fait que les microarchitectures modernes comportent de nombreux mécanismes d'optimisation des performances moyennes, du fait par ailleurs de l'absence d'une notion de temps explicite dans les programmes, auxquels s'ajoutent lors d'une exécution l'indétermination des données d'entrée et de l'état initial du matériel, il existe une importante dispersion des temps d'exécution possibles d'un même programme sur une même microarchitecture.

Ainsi, le comportement des systèmes temps-réel doit être validé par des analyses temporelles rigoureuses, généralement destinées à déterminer le pire temps d'exécution d'un programme sur une cible matérielle. Il existe plusieurs méthodes pour conduire de telles analyses, parmi lesquelles les méthodes génériques d'analyse statique ; on peut aussi citer les méthodes basées sur des mesures ou des analyses probabilistes. Toutes ces méthodes sont pourtant mises en péril par des phénomènes d'exécution nommés *anomalies temporelles* (AT). Une AT se manifeste au niveau d'au moins deux traces d'exécution du même programme avec les mêmes données d'entrée, néanmoins à partir d'états initiaux matériels distincts qui provoquent localement une variation de *latence*, par exemple avec un défaut de cache (*cache miss*) pour une instruction dans une trace alors que la même instruction connaît un succès de cache (*cache hit*) dans l'autre trace. Nous distinguons deux types d'AT : les AT contre-intuitives et les AT d'amplification. Une AT *contre-intuitive* a lieu quand une accélération locale (succès de cache par exemple) conduit à un ralentissement global, tandis qu'une AT d'*amplification* se produit quand un ralentissement local (défaut de cache par exemple) entraîne un ralentissement global encore plus grand.

La compréhension et la détection des AT sont donc cruciales pour l'analyse des systèmes temps-réel. Étant donné que de plus en plus de processeurs standards sont utilisés dans les systèmes temps-réels et que ces processeurs sont connus, par leurs nombreux mécanismes d'optimisation, pour présenter des AT, il en est d'autant plus important de mettre en œuvre des *procédures de détection* fiables. Nous détaillons dans ce manuscrit les domaines de l'état de l'art qui y concourent : la modélisation du temps dans les microarchitectures, l'utilisation des méthodes

formelles pour la vérification matérielle, la littérature portant sur les outils de modélisation et vérification formelles, et enfin nous mettons l'accent sur la formalisation des AT au niveau de la microarchitecture. En particulier, nous établissons une classification des différentes définitions préexistantes des AT contre-intuitives, c'est-à-dire les AT les plus documentées, selon plusieurs critères :

- le critère utilisé pour définir (formellement) les latences et donc les *variations* ;
- l'*interprétation* du phénomène qui est faite vis-à-vis de ses conséquences potentielles sur les analyses pire temps ;
- l'existence d'une définition complémentaire pour les AT d'amplification ;
- et enfin l'existence d'une procédure de détection *implémentée* associée aux notions théoriques exposées.

Nous montrons ainsi que les diverses définitions sont essentiellement différentes et donc qu'elles nécessitent d'être approfondies et comparées, mais aussi que la plupart des travaux existants demeurent théoriques et sans applications concrètes – d'où la nécessité de proposer des procédures de détection.

Alors que la présence d'AT contre-intuitives menace le bien-fondé ou la flexibilité des analyses, il n'existe pas d'outils pour les détecter de manière systématique. En outre, leurs définitions formelles sont souvent incomplètes et illustrées seulement à travers des exemples partiels. Nous proposons une structure formelle unifiée pour évaluer les définitions existantes des AT à partir d'un modèle générique de pipeline out-of-order. Les principales contributions sont alors :

- une approche *systématique*, avec des hypothèses précises ;
- un modèle formel concret du matériel (spécifié en TLA^+) ;
- une *évaluation comparative* des différentes définitions par vérification de modèle (*model checking*).

Cela nous permet de montrer qu'aucune définition ne domine les autres ni n'est suffisamment précise pour détecter les AT de manière fiable. Nous montrons de plus que le principal défaut expliquant l'absence d'outils est l'absence de relations entre les effets temporels locaux et globaux. Nous y répondons par le concept important de *causalité*, que nous utilisons pour revoir la formalisation de ces AT. Nous proposons systématiquement des notions génériques, mais aussi des instances de ces notions spécialisées pour les pipelines out-of-order. Nous évaluons notre procédure de détection subséquente sur des exemples illustratifs et bancs de tests, montrant qu'elle permet de capturer précisément les AT.

Par ailleurs, l'inhérente complexité des systèmes exige que leurs analyses gèrent l'important espace d'états résultant. Une solution est de réaliser des analyses *compositionnelles*, dans lesquelles le comportement temporel du système est obtenu à

partir des contributions individuelles de différents composants. Ces analyses compositionnelles sont précisément menacées par les AT d'amplification. Nous faisons progresser l'étude de ces AT en montrant comment une abstraction spécialisée peut être adaptée pour analyser les AT d'amplification de manière efficace sur un processeur industriel, en l'occurence le processeur TriCore d'Infineon, largement utilisé dans l'industrie automobile. Par rapport aux processeurs prédictibles sur lesquels cette abstraction avait déjà été appliquée, la microarchitecture beaucoup plus complexe (double pipeline et hiérarchie mémoire) nécessite un passage à l'échelle, tant des fonctionnalités modélisées que de l'espace d'états. De plus, ce processeur industriel – standard – n'est pas conçu comme un processeur prédictible et est donc susceptible de donner lieu à des AT : la problématique n'est plus de prouver l'absence d'AT mais plutôt d'explorer les différentes sources possibles de variations qui conduisent à des AT. Les principales contributions sont alors :

- des extensions structurelles et fonctionnelles de l'abstraction spécialisée pour notre cas d'étude ;
- des réductions systématiques consistant à supprimer les configurations inutiles de l'espace d'états, en prenant en compte les spécificités de la microarchitecture (effets superscalaires notamment) ;
- une évaluation de la complexité de notre procédure ;
- une illustration, à partir de cette classe d'AT, de la façon dont des méthodes de vérification fondées sur les contre-exemples peuvent être utilisées en vue de l'identification de motifs logiciels conduisant potentiellement à des AT lors de l'exécution (pour, à plus long terme, fournir des contre-mesures).

Enfin, nous fournissons des pistes de réflexion sur les fortes interactions potentielles et bidirectionnelles entre les deux branches de nos travaux, à savoir la formalisation précise des concepts à partir des AT contre-intuitives (les plus documentées) et les stratégies de vérification en vue de mettre en œuvre des contre-mesures, à partir de l'exemple des AT d'amplification (sur la base de travaux préalables supposant une condition nécessaire).

INTRODUCTION

REAL-TIME systems are subjected to timing requirements, which demands that they be *predictable*. That means that their timing behavior must be soundly estimable offline, before they are operational. *Timing Anomalies* (TAs) are execution phenomena known to hinder the predictability of real-time systems. In an ideal situation, real-time applications should thus rely on predictable, TA-free architectures. However, COTS (Commercial Off-The-Shelf) processors are more and more adopted for real-time applications, in order to reduce costs and, in mixed-criticality systems, to benefit from their performance enhancers. These enhancers are likely to introduce TAs and to jeopardize predictability. Hence, it is fundamental to reliably detect TAs in real-time systems. Only formal verification can provide safe guarantees on the absence of TAs or identify their occurrences accurately. Since real-time systems run application-specific software, and analyses are conducted with regard to the executed software, we must formally verify that the execution of a given program is free from TAs or, in default thereof, identify TAs accurately. In this code-specific approach, the system is seen as a combination of both hardware and software components. We thus need formal models that integrate both aspects. **The first part of this thesis elaborates on this context and the microarchitecture case studies, before setting up the formal-verification framework and presenting the related work.** We introduce two types of TAs, namely *counter-intuitive* TAs, which are more documented, and *amplification* TAs. Right after this part, **we state more specifically the problems that we address in the next parts of the thesis**.

A natural step in achieving a reliable detection of TAs consists in disposing of a formal definition of the phenomenon. Yet, the usual interpretation of the term *Timing Anomaly* remains rather colloquial and the understanding of the underlying effects is often only illustrated through simple examples that give some *intuitive* understanding. Existing work often provides abstract notions, making it difficult to apply the definitions—let alone reason about TAs—on concrete applications. **In the second part of this thesis, we develop a formal model of an out-of-order pipeline [1] and we peruse the existing definitions of counter-intuitive TAs in light of this concrete hardware model, exhibiting their limitations [2].** From the outcome of this work, we propose a *precise* and *applicable* formal definition of counter-intuitive TAs. From this definition, we implement a TA-*detection procedure* for real-time systems. **In the third part of this thesis, we present our novel definition of counter-intuitive TAs and the related detection procedure [3].**

Regardless of the underlying formal definitions of TAs, we have investigated how architectures can be appropriately modeled to verify timing properties of pro-

gram executions, and how formal tools can be harnessed to derive TA patterns, for the purpose of inserting counter-measures. This work concerns in the first instance amplification TAs, for which no precise detection procedure exists—only heuristics are known to detect them—, but it could also concern counter-intuitive TAs, for which our novel detection procedure can be utilized—the procedure alone is limited to a verdict and, potentially, a single counterexample showing a TA. **In the fourth part of this thesis, which is independent of the previous two, we illustrate the detection of amplification TAs on an industrial case study [4], and we provide generic heuristics to derive TA patterns [5].**

The last chapter of this document concludes the thesis and outlines future work.

CONTRIBUTIONS

[1] Benjamin Binder et al. “Formal Processor Modeling for Analyzing Safety and Security Properties”. In: *11th European Congress Embedded Real Time Systems (ERTS)*. 2022.

[2] Benjamin Binder et al. “Is This Still Normal? Putting Definitions of Timing Anomalies to the Test”. In: *IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2021, pp. 139–148. DOI: 10.1109/RTCSA52859.2021.00024.

[3] Benjamin Binder et al. “The Role of Causality in a Formal Definition of Timing Anomalies”. In: *IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2022, pp. 91–102. DOI: 10.1109/RTCSA55878.2022.00016.

[4] Benjamin Binder et al. “Scalable Detection of Amplification Timing Anomalies for the Superscalar TriCore Architecture”. In: *Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings*. Vol. 12327. Lecture Notes in Computer Science. Springer, 2020, pp. 151–169. DOI: 10.1007/978-3-030-58298-2_6.

[5] Benjamin Binder et al. “Formal Modeling and Verification for Amplification Timing Anomalies in the Superscalar TriCore Architecture”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 24 (2022), pp. 415–440. ISSN: 1433-2787. DOI: 10.1007/s10009-022-00655-1.

CONTENTS

I Background 17

1 Context 21

2 Microarchitecture Case Studies 33

3 Formal Verification 45

4 Related Work 59

Problem Statements 79

II Limitations of the Existing Definitions of Counter-Intuitive Timing Anomalies 81

5 Interpretation and Modeling of the Definitions 85

6 Assessment of the Definitions 103

III Detection of Counter-Intuitive Timing Anomalies 113

7 A Novel Formal Definition 117

8 Detection Procedure 139

IV Heuristics for the Detection of Timing-Anomaly Patterns 161

9 Detection of Amplification TAs 165

10 Towards Software-Related Patterns 185

Conclusion & Prospects 201

Bibliography 203

Part I
Background

NEXT, we introduce the background on which this thesis stems. We present the context within the thesis falls, the hardware case studies that we have retained, the formal methods and tools that we used and the related work. All these elements allow us to accurately state the problems that we outlined in the introduction and on which we elaborate in the next parts of the thesis.

CONTENTS

1 Context . . . 21
1.1 Real-Time Systems . . . 21
1.1.1 Microprocessors . . . 21
1.1.2 From Embedded Systems to Safety-Critical Systems . . . 22
1.1.3 Predictability . . . 23
1.1.4 Timing Analysis . . . 24
1.2 Pipelines . . . 27
1.2.1 Principle . . . 27
1.2.2 Hazards and Stalling, Static/Dynamic Scheduling . . . 27
1.2.3 Single-Issue/Superscalar Pipelines . . . 28
1.3 Timing Anomalies in Microarchitectures . . . 29
1.3.1 Intuitive Definitions . . . 29
1.3.2 Timing Anomalies against Timing Analysis . . . 31
1.4 Summary: Context of the Thesis . . . 31
2 Microarchitecture Case Studies . . . 33
2.1 Classical In-Order Pipeline . . . 33
2.1.1 Description of the Pipeline . . . 33
2.1.2 Guideline Example of Amplification TAs . . . 35
2.2 Overview of the TriCore Microarchitecture . . . 36
2.2.1 The TriCore Microarchitecture . . . 36
2.2.2 Timing Behavior . . . 37
2.3 A Representative Out-of-Order-Pipeline Template. . . . 41
2.3.1 Pipeline Overview . . . 41
2.3.2 Execution Functioning . . . 42
2.3.3 Traditional Pattern of Counter-Intuitive TAs . . . 43

3 Formal Verification . . . 45
3.1 Formal Notions . . . 46
3.2 Model Checking . . . 49
3.2.1 Invariants . . . 49
3.2.2 Explicit Model Checking . . . 50
3.2.3 Symbolic Model Checking . . . 51
3.2.4 Counterexample-Guided Methods . . . 52
3.3 Modeling and Verification Tools . . . 52
3.3.1 UCLID5 . . . 53
3.3.2 TLA$^+$ . . . 54
3.4 Applications of Formal Verification . . . 56
3.5 Summary: our Formal Framework. . . . 57
4 Related Work . . . 59
4.1 Interpretations of Counter-Intuitive TAs . . . 59
4.1.1 Concrete and Abstract Models . . . 59
4.1.2 Static-Analysis-Centric Interpretation . . . 61
4.1.3 Hardware-Centric Interpretation . . . 63
4.1.4 Absolute-WCET Interpretation vs. Pairwise Interpretation 63
4.2 Overview of the Definitions of TAs . . . 65
4.2.1 Step Heights in Step Functions . . . 66
4.2.2 Intersections in Step Functions . . . 68
4.2.3 Component Occupation . . . 69
4.2.4 Instruction Locality . . . 69
4.3 Predictable Cores . . . 70
4.3.1 Specific Hardware Designs . . . 71
4.3.2 Canonical Model for Assessing Compositionality . . . 72
4.4 Timing Modeling in Pipelines . . . 73
4.4.1 Model Checking of Timing Properties . . . 74
4.4.2 Analytical Methods . . . 75
4.4.3 Timing Modeling of TriCore . . . 75
4.5 Summary: the Definitions of Timing Anomalies . . . 76

1 – CONTEXT

In this chapter, we provide a general background on microprocessor-based systems and those qualified as real-time systems in particular. First, we elaborate on general microarchitecture notions, as well as on important notions specific to these systems, such as predictability and timing analysis (Sec. 1.1). Then, we introduce the main notions related to microarchitectural pipelines (Sec. 1.2), before describing the phenomena called *Timing Anomalies* (TAs) (Sec. 1.3).

1.1 . Real-Time Systems

We first describe generic microprocessor-based computational systems, which are made of hardware and software components (Sec. 1.1.1). Then, we focus on real-time systems, which are subjected to timing constraints that are explicitly part of their specification (Sec. 1.1.2). Consequently, verifying real-time systems requires checking that these constraints are satisfied. The ability of these systems to be analyzed in a timing-oriented approach is called (timing) *predictability* (Sec. 1.1.3). Finally, we introduce generalities on timing analysis and a common metrics, the *Worst-Case Execution Time* (WCET) (Sec. 1.1.4).

1.1.1 . Microprocessors

Digital systems are made of integrated circuits, i.e., electronic assemblies that use digital signals to encode information. An important class of digital systems is *processors*. They are specific circuits designed to execute *programs*, i.e., sequences of instructions. Processors contained in a single integrated circuit are named microprocessors. In the minimal configuration, actual systems connect a microprocessor with one or several memories through a bus. Memories contain the instructions and the data manipulated by the program. A microprocessor design can conceptually be decomposed into a control path and a data path. The main component of the data path is the Arithmetic Logic Unit (ALU), i.e., a dedicated hardware for integer operations. The main component of the control path is the (logic) controller, which orchestrates the execution of the program, by monitoring the state of the data path. It is responsible for retrieving the instructions of the program from memory and for adequately configuring the data path (by means of multiplexers), so that the ALU performs the operations actually required by the instructions.

Processors have the advantage over other classes of integrated circuits that they can be easily programmed and reprogrammed. For these operations, the hardware design is totally unchanged. Only the sequence of instructions is loaded into memory. Consequently, the execution of a program on a processor can be seen as the combination of two independent subsystems: a *hardware* system—the fixed processor design, also known as microarchitecture—and a *software* system—the

program that is intended to condition the behavior of the processor on the fly, by imposing its semantics to the hardware. An important effort has to be made to write programs, which can be written in general-purpose programming languages. Programs are to be compiled and built and, ultimately, loaded into memory in the form of machine instructions. The instructions that a processor is able to execute form its *Instruction Set Architecture* (ISA). Assembly languages allow an intermediate, human-readable representation of machine instructions; they thus constitute an interface with hardware. Since processors are exclusively designed to execute programs, the ISA constitutes the *functional specification* of a processor, whereas the concrete hardware design of the processor is an *implementation* of the ISA. **In this document, we restrict the notion of hardware to that of microprocessor microarchitecture, and the notion of software to assembly programs.**

Finally, processors may contain several cores with shared resources, so as to perform a parallel execution of some instructions. The assignment of tasks to the various cores and to the shared resources is out of the scope of this thesis. We will focus on the consequences induced by shared resources at a lower level, namely the pipeline level (Sec. 1.2).

1.1.2 . From Embedded Systems to Safety-Critical Systems

Processors are intensively used in purely computational systems, such as desktop computers or servers. However, they are also found in almost all ubiquitous, daily systems. In this case, the computational subsystem is called an *embedded system*, and it is dedicated to a specialized application. The embedded system refers to the software components as well as to the underlying microarchitecture. Whereas all computational systems must comply with functional requirements, which specify what they must do, the environment of embedded systems subjects them to additional, non-functional requirements that specify *under which constraints*, e.g., regarding weight, size, temperature, or timing. Embedded systems are sometimes called Cyber-Physical Systems (CPS), but this terminology notably refers to complex networked embedded systems [6], in the context of the Fourth Industrial Revolution and, in particular, the Internet of Things (IoT). **We focus on (traditional microprocessor-based) embedded systems, and the non-functional characteristic of interest is *time*.**

Some embedded systems are designed to maintain a permanent interaction with their environment, by producing actions/physical outputs in response to stimuli/physical inputs, through actuators and sensors. An important subset of reactive systems is that of *Real-Time* (RT) systems, which are moreover subjected to *external timing constraints* [7, 8]. **In this thesis, we focus on real-time (reactive embedded) systems.** Real-time systems can be divided into soft, firm, and hard real-time systems, depending on the fixed tolerance regarding the deadlines imposed by the timing constraints. In soft real time, deadline misses are allowed and results remain usable to a certain extent after the deadlines. In firm real time,

sparse deadline misses are tolerable but associated to outdated, unused values. Finally, deadline misses are considered failures for hard real-time systems. When failures are likely to entail severe consequences (on equipment, environment, or human beings), such systems are qualified as safety-critical (e.g., the Flight Management System of an aircraft). Obviously, providing guarantees regarding the timing constraints is all the more relevant for this class of systems.

1.1.3 . Predictability

In the context of real-time systems, the correctness of the execution of a program on a given microarchitecture relies as much on its temporal behavior as on its functional aspects. Hard real-time systems are often subjected to certification. Their timing behavior is to be verified as well as their functional correctness. Hence, executions, and thus underlying microarchitectures, should ideally be fully *predictable*. **Timing predictability refers to the ability of computational systems (or resulting executions) to be analyzed in order to derive safe guarantees on the timing behavior, before any actual execution.** The *raison d'être* of this problem—estimating time—stems from several causes:

1. Programs do not embed any notion of physical, microarchitectural time, but only a notion of logical, sequential time through the instruction order—hence, analyzing the timing behavior is important.

2. Modern microarchitectures contain performance enhancers that may perform the same computations in different ways, depending on the values of the instruction operands and on the hardware state—we face a finite set of time values that may be characterized by its bounds.

3. The input data and the complete initial state of the microarchitecture are not known in advance—we can only compute *estimates*.

In practice, predictability is not a clear-cut property; a microarchitecture can be deemed predictable as it implements features that are intended to ease analyses and as it tends to enforce a regular timing behavior in certain situations. We will elaborate on predictable microarchitectures in Sec. 4.3.

Real-time systems are subjected to external timing constraints; hence, transposing the logical time of the program onto the physical time of the environment is crucial (item 1). Processors rely on sequential circuits triggered by specific signals called clocks. They do not execute instructions in zero time; each instruction requires a certain number of clock cycles, primarily depending on the instruction type. Clock rate (or frequency) f is in turn limited by the design of the data path and the physical features of its digital components—the clock cycle $1/f$ is limited by the *critical path*, i.e., the chain of combinatorial components between an input and an output of the data path that has the largest delay. As a consequence, the implementation imposes a certain time T to execute a sequence of N instructions,

which can be approximated by the basic performance equation through the average clock-Cycle-Per-Instruction indicator CPI [9]:

$$T \approx N \times \mathrm{CPI} \times \frac{1}{f} \tag{1.1}$$

Performance enhancers (item 2), such as caches and predictors, are complex mechanisms that aim at improving the average execution time of a program. These mechanisms may store reusable data or implement specific treatments for certain operand values to speed computations up. A cache memory stores a portion of the main-memory content on a fast chip. When the processor executes a read-memory instruction, it first checks whether the data are available in the cache. If so, this is a *cache hit* and the processor gets the data directly from the cache; otherwise, this is a *cache miss* and the data are loaded from the main memory, through the memory bus, and then stored in the cache (potentially replacing another value according to a policy). In the favorable cases (e.g., cache hits), on which all overall optimization mechanisms bet, the speedup may be considerable; however, performance enhancers introduce an important microarchitectural-state, and thus *time* dispersion. This dispersion runs counter to predictability, since the number of clock cycles to execute one instruction may strongly vary (even for the same instruction class or the same instruction), making the average number CPI (cf. Eq. 1.1) a poor indicator, against the bounds notably.

Finally (item 3), it is important to keep in mind that processors, as all traditional computational systems, are deterministic. This means that for a given input, the initial state fully determines the whole execution. Thus, in spite of the time dispersion that specific mechanisms may introduce, it is theoretically possible to accurately compute the time required to execute a program, by determining and accumulating the number of clock cycles per individual instruction (in Eq. 1.1)—depending on the initial processor state, e.g., the contents of the cache memories. The problem resides, on the one hand, in the undetermined initial state in the analysis stage, and on the other hand, in the complexity of microarchitectures and the huge resulting state space that prevent from performing an exhaustive exploration in practice, due to state explosion. Hence, one must compute estimates, of the timing bounds in particular.

1.1.4 . Timing Analysis

Studying predictability means analyzing the distribution of the *possible execution times* of the application program on the microarchitecture of the real-time system (see Fig. 1.1). All possible executions—depending on the initial hardware state and the input data—form the *actual* distribution of times, i.e., the top, dark curve on Fig. 1.1. The longest execution time is called the *Worst-Case Execution Time* (WCET) and the shortest execution time is called the Best-Case Execution Time (BCET). As explained above, the hardware state and the input data that lead to these two particular execution times are not known. *Timing analysis* is often

limited to the estimation of these timing bounds.

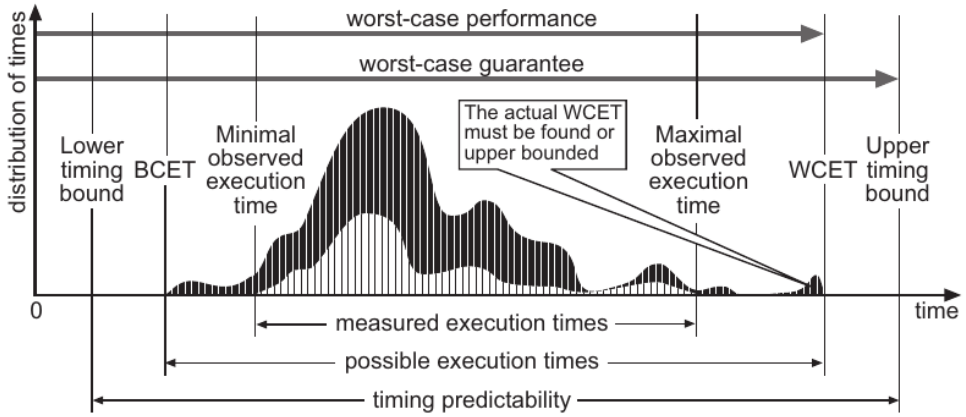


Figure 1.1: Distribution of execution times of a program (or a task) on a processor, Worst-Case Execution Time (WCET), and estimations (from [10]).

We focus on *Worst-Case-Execution-Time* (WCET) *analysis*. Estimating the WCET is of particular importance in order to provide guarantees on the timing correctness. Critical real-time systems are often statically scheduled, namely timing quanta are allocated to the various program tasks for them to execute on the processor. The preallocated durations are in turn subordinated to the automatic-control laws that produce physical outputs in reaction to physical stimuli. As a consequence, the execution time of a task should never exceed the preallocated time that enforces deadlines; in other words, the WCET should be shorter than the preallocated time. In dynamically scheduled systems, the task deadlines are determined on the fly, according to specific algorithms. In this case, schedulability analyses must guarantee that all tasks will execute in time. Again, WCET estimation is important, since the task WCETs are an essential input of these analyses.

Several methods exist to estimate the WCET of a task. Some are based on test measurements [11, 12] or probabilistic analysis [13, 14, 15]. However, these approaches alone are obviously non-exhaustive and thus are not safe to provide timing guarantees. Fig. 1.1 illustrates that only a subset of the possible execution times can be observed, which constitutes the distribution of *measured execution times*. From this distribution can be derived the minimal and the *maximal observed execution time*. This latter value is a WCET estimate that is likely to be underestimated (as a bound of a subset of the possible execution times).

So as to provide safe guarantees, analyses must cope with the absence of exhaustivity. Specific methods, grouped together under the name of *static analysis* [10, 16], allow substituting the intractable exhaustive exploration of the concrete hardware by operations on an abstract microarchitectural model. They use abstractions to compute an over-approximation of the possible hardware states that may

appear during the program execution. It is again possible to derive the longest execution time from this abstract model, which constitutes another WCET estimate. By abuse of language, this estimate from static analysis is sometimes referred to as *the* WCET [10], notably owing to the fact that the *actual* WCET is inaccessible. We will try to systematically use the phrase "WCET estimate" when we refer to an estimate. If the system is *predictable* (cf. Sec. 1.1.3), this estimate is supposed to be safe, i.e., an overestimation of the actual value (see the upper timing bound for predictability in Fig. 1.1). However, the estimation is more or less tight, i.e., the WCET is more or less overestimated, depending both on the concrete and the related abstract models [10].

The static WCET analysis of a task lies on three main steps [10, 17, 16]:

1. program, control-flow analysis;
2. **processor-behavior, microarchitectural analysis**;
3. global bound calculation (or path analysis).

First (item 1), programs may contain branches that influence the executed instruction sequence, depending on the input data in particular. Thus, besides the time dispersion introduced by hardware, the program itself may also allow various (functional or non-functional) behaviors. Control-flow analysis is required to find out all feasible program paths. Moreover, programs may contain loops, whose bounds could depend on the input data and must be determined. In this thesis, we consider unrolled sequences of instructions with fixed input data, thus one fixed program path at a time. This first step identifies *basic blocks*, which are considered atomic in the analysis, namely the instructions of the basic blocs are executed one after another, in the same order and without any interruption, in any execution of the program.

At this level, a microarchitectural analysis (item 2) is required to determine the time needed to execute the program on the processor—this is the shift from the code-level time (in terms of executed instructions) to the microarchitecture-level time (in clock cycles), mentioned in Sec. 1.1.3. When mentioning static WCET analysis in the remainder, we focus on this second step of the standard analysis. Finally, analytical methods, such as the *Implicit Path Enumeration Technique* (IPET) [10], are used to compute global bounds.

Lastly (item 3), the complexity of microarchitectures makes it necessary to perform the microarchitectural analysis in a compositional way [18, 19], by adding up the timing contributions of individual microarchitecture elements. A compositional analysis, for instance, would consider separate microarchitectural timing analyses for caches and pipelines and then combine their respective timing results.

1.2 . Pipelines

All modern processor cores integrate hardware *pipelining*, a basic feature to improve execution throughput. Yet, pipelines highly complicate timing analyses, since the execution of one instruction depends on the execution of other instructions that simultaneously make use of shared hardware resources. We now explain the general functioning of pipelines (Sec. 1.2.1), the notion of (pipeline) hazards and their consequences (Sec. 1.2.2), and the notion of superscalarity (Sec. 1.2.3).

1.2.1 . Principle

Pipelines work on the principle that the execution of an instruction can be divided into several steps relying on different elements of the microarchitecture. Thus, they implement Instruction-Level Parallelism (ILP) by processing several instructions simultaneously [9], in distinct parts of the microarchitecture called *pipeline stages*.

Pipelines make it possible to reach the ideal throughput of 1 instruction per cycle, since in steady state (when the pipeline is filled), one instruction may complete and leave the pipeline in each cycle. In this case, the average number of cycles per instruction is CPI $= 1$ (cf. Eq. 1.1), so that the pipelined-execution time $T_{\text{ideal pipeline}}$ of a sequence of N instructions is:

$$T_{\text{ideal pipeline}} = N \times \frac{1}{f} \tag{1.2}$$

Note that the resulting execution time could certainly not be achieved without pipelining, since it would require a very complex data path to fully execute all instructions in a single cycle (i.e., the ideal throughput), which moreover would drastically lengthen the critical path and limit the maximal frequency (see Sec. 1.1.3). However, the ideal throughput of a pipelined execution is maintained only if one instruction completes in each cycle. This is possible only if the instruction flow through the successive pipeline stages is uninterrupted. Yet, besides the penalties introduced by performance enhancers in unfavorable cases (e.g., cache misses), **the pipeline must be *stalled* in some situations called (pipeline) *hazards***. In these cases, the incriminated instructions—as well as other instructions, according to the stalling logic—occupy certain pipeline stages but do not perform any computation, so that a *bubble* is inserted in the pipeline.

1.2.2 . Hazards and Stalling, Static/Dynamic Scheduling

There exist three types of hazards: *structural*, *data*, and *control* hazards [9]. Structural hazards occur when two instructions in the pipeline need the same hardware resource in the same cycle, e.g., the memory bus. Data hazards are due to data dependencies in the program, when an instruction depends on the result of a previous instruction that has not been produced yet. Finally, control hazards occur when the next value of the Program Counter (PC), i.e., the address of the next instruction to be executed, is not known (or is mispredicted if the microarchitecture is endowed with speculation mechanisms); the (actual) PC following a control instruction is not known until this instruction computes whether a branch should

be taken or not. The timing contribution of all stall cycles must be accounted for, which entails an effective throughput lower than the ideal 1 instruction per cycle.

Data dependencies are common in programs. The data hazards that they may entail can be reduced by implementing mainly the two following mechanisms (possibly together): *forwarding/bypassing* and *Out-of-Order (OoO) execution/dynamic scheduling* [9]. Forwarding extends the data path with signals that communicate the results of the computations produced by an instruction in one stage to previous stages (processing younger instructions), thus propagating the computed results at the soonest for dependent instructions in the pipeline. **Forwarding and OoO execution tend to maintain the throughput as close as possible to the ideal throughput.**

Pipelines always fetch instructions in program order, according to the sequential semantics of programs. Whereas *in-order/statically scheduled* pipelines preserve this order through each pipeline stage, OoO execution allows the scheduling of instructions to the functional units (where the actual computations are made on the operands) in a different order from the program one. Obviously, specific mechanisms must ensure that OoO execution respects data dependencies. All modern OoO microarchitectures also implement a reorder buffer (ROB) so as to commit instructions in program order, so that the effective processor state is updated in program order and the OoO engine is transparent to the user.

OoO execution means dynamic scheduling, and thus requires a dynamic issue structure to detect hazards, through dedicated hardware components. However, note that even in-order pipelines must issue instructions that comply with data hazards, so as to respect data dependencies. The issue structure of in-order pipelines may be either static (with a hazard detection at the software level) or dynamic (hazard detection at the hardware, pipeline level).

1.2.3 . Single-Issue/Superscalar Pipelines

Pipelines are characterized by their length/depth, i.e., the number of pipeline stages, which results from a trade-off between the microarchitectural complexity (including hazard management) and the maximal frequency (as evoked above in Sec. 1.1.3). They are also characterized by the number of copies of the same pipeline stages or similar resources (e.g., functional units) in the pipeline. Pipelines with a single set of such resources are called *single-issue* (or *scalar*). They contrast to *multiple-issue* pipelines [9], which allow several instructions to be issued to the functional units in the same cycle, so as to keep all of the functional units busy. All other pipeline stages are often multiplied consequently, in order to benefit fully and at lower cost from the multiple issue, raising the ideal throughput beyond 1 instruction per cycle.

Multiple-issue pipelines benefit even more than single-issue pipelines from OoO execution (since the instructions issued at the same time may execute in parallel), which leads to a specific classification of multiple-issue pipelines under their (static/dynamic) scheduling features and the associated issue structures. Multiple-issue

pipelines with dynamic issue structures are called *superscalar*. These pipelines issue a varying number of instructions per cycle, depending on the hazard detection, handled by hardware. They contrast, notably, with VLIW (Very Long Instruction Word) pipelines, which issue a fixed number of instructions statically gathered by the compiler into bundles. **In the remainder, we study various real-time systems, containing a single-issue or superscalar, in-order or OoO pipeline microarchitecture.**

1.3 . Timing Anomalies in Microarchitectures

We mentioned that performance enhancers are complex mechanisms that introduce many microarchitecture states (cf. Sec. 1.1.3). None of the timing-analysis methods is able to explore all possible executions in the concrete hardware. They thus only provide safe WCET estimates (cf. Sec. 1.1.4) when certain underlying hypotheses (Sec. 1.3.2) are satisfied. Yet, undesired timing phenomena, called *Timing Anomalies* (TAs), can manifest, threatening these hypotheses and the soundness of timing analyses.

In this section, we first provide an intuitive definition of two classes of TAs, specifically counter-intuitive and amplification TAs, and a brief historic background (Sec. 1.3.1). Then, we explain to what extent TAs are problematic for timing analyses (Sec. 1.3.2).

1.3.1 . Intuitive Definitions

Intuitively, a TA is a local condition at a given moment during the execution of a real-time software that leads to an *undesired* effect on the (global) execution time. TAs manifest at the level of (at least) two different *execution* traces corresponding to the same program trace, i.e., with the *same* input program and data, yet starting from distinct initial hardware states. **A *counter-intuitive* TA occurs when a local speedup of one trace (wrt. another one), e.g., a cache hit (instead of a miss in the other trace), leads to a larger global execution time. An *amplification* TA occurs when a local slowdown of one trace, e.g., a cache miss (instead of a hit in another trace), leads to an even larger global execution time.** Both of these TA variants may also be described in the opposite point of view, where the sense of the local variation is reversed and the execution time is reduced. The point of view where the execution time is increased is generally preferred, since these effects may have an impact on the actual WCET and thus have to be considered during timing analysis.

Fig. 1.2 exemplifies TAs by mapping a reference execution (Execution 1), as well as various possible executions (Executions 2a/b/c) of the same program induced by the variation of a local execution variation (in gray). Compared to Execution 2a, the local speedup (Δ_L) of Execution 1 leads to an increased global execution time (Δ_G^{CI}), thus Execution 1 shows a counter-intuitive TA wrt. Execution 2a. Compared to Execution 1, the local slowdown (Δ_L) of Execution 2b leads

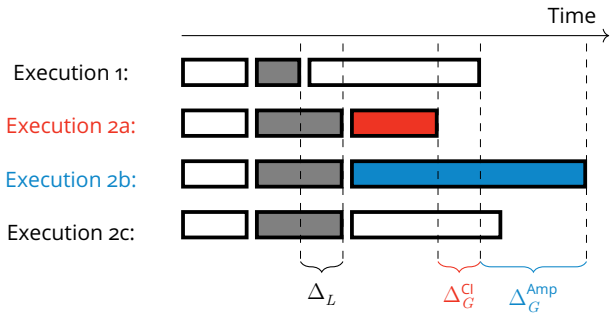


Figure 1.2: Schematic of timing anomalies (TAs) from various executions of the same program with a local timing variation Δ_L: Executions 1 and 2a show a counter-intuitive (CI) TA, whereas Executions 1 and 2b show an amplification (Amp) TA. Execution 2c shows no TA.

to a proportionally larger increase ($\Delta_G^{\text{Amp}} > \Delta_L$) of the global execution time, thus Execution 2b shows an amplification TA wrt. Execution 1. In contrast, the local slowdown (Δ_L) of Execution 2c compared to Execution 1 leads to an increased global execution time, and this global increase is smaller than Δ_L. Thus, Execution 2c shows no TA.

In practice, local variations may have various root causes: notably, variable instruction latency (e.g., for division or multiplication) or variable memory-access time (e.g., due to DRAM page conflicts or cache misses). Microarchitecture features like out-of-order execution [20, 21, 22, 23, 24], specific cache policies [22, 24], branch prediction [22, 24], or contention of memory accesses [19, 25, 26] (e.g., because of the bus arbitration) may lead to TAs. TAs caused by OoO execution, called *scheduling anomalies*, are the most documented anomalies [ibid.], and, besides, counter-intuitive TAs are systematically illustrated from the scheduling on the functional units of an OoO pipeline (cf. Sec. 2.3.3). To our knowledge, contention of memory accesses is specific to amplification TAs, and such TAs may occur even in simple in-order pipelines (cf. Sec. 2.1.2) and were recently studied [ibid.]. **Thus, we will illustrate counter-intuitive TAs through scheduling TAs and amplification TAs through TAs caused by contention of memory accesses, and, without restricting the generality, we will often interpret local timing variations as cache misses** (see the case studies introduced in Ch. 2).

Counter-intuitive TAs were first studied, back in the sixties, in the context of task scheduling on uniform multi-processors [27, 28, 29, 30, 31], which differs from our work in several aspects. First, the granularity of tasks erases the presence of an executed code. Second, resources are considered to be identical processors in the context of tasks, whereas we have specialized pipeline resources. Third, all parameters are controllable inputs of scheduling algorithms, whereas we consider

executions on a fixed hardware microarchitecture. Notably, such algorithms may consider task priorities (nonexistent in our context) or the partial order between tasks—we have a fixed total order due to the instruction order (for in-order microarchitectures) or partial order due to the data dependencies in the program (for OoO microarchitectures).

1.3.2 . Timing Anomalies against Timing Analysis

Counter-intuitive TAs pose a challenge to all WCET approaches, due to their impact on their hypotheses (cf. Sec. 1.1.4). In particular for static analyses, counter-intuitive TAs are problematic because it is generally no longer possible to consider only the local worst-case variations [10] (see Executions 2a against 1 in Fig. 1.2). Instead, an *exhaustive exploration* of the reachable hardware states has to be performed, which is costly or often even prohibitive. The same issue arises for test-based approaches relying on measurements, since the possible number of tests to cover increases drastically. Probabilistic methods are also jeopardized. Slight changes in the hardware state may trigger a counter-intuitive TA, which in turn may cause a considerable increase of the execution time. TAs may thus invalidate fundamental hypotheses of probabilistic approaches (e.g., independence and stationarity [15]) and thus pose a threat to the validity of the obtained results.

Amplification TAs hinder compositional timing analysis [19, 26]. The complexity of modern microarchitectures makes it necessary to perform timing analysis in a compositional way (cf. Sec. 1.1.4). In the presence of such TAs and in order to perform a sound compositional timing analysis, it becomes essential to bound the amplification effects. Note that counter-intuitive TAs jeopardize timing analysis in general, thus also compositional analysis in particular.

1.4 . Summary: Context of the Thesis

We introduced general notions about real-time systems and computer microarchitectures that are used all along this document. In the next chapters, we focus on real-time systems whose hardware microarchitectures are comprised of a pipelined microprocessor, including caches and connected to a memory system. We then consider pipelined executions resulting from the execution of (low-level) programs on these microarchitectures. Hazards entail stalling in pipelines, thus deviating from the ideal timing behavior. Forwarding and Out-of-Order (OoO) execution are specific mechanisms that allow reducing the occurrence of hazards and getting close to the ideal pipeline throughput.

We intend to study the timing predictability of real-time systems, namely to what extent they can be analyzed off-line to derive timing guarantees, focusing on the Worst-Case Execution Time (WCET). Forwarding, OoO execution, and superscalar features are likely to trigger variations in different execution traces of the same program, which in turn may trigger *counter-intuitive* or *amplification* timing anomalies (TAs). Counter-intuitive TAs pose a challenge to all WCET-

analysis methods and amplification TAs prevent compositional timing analyses.

A counter-intuitive TA occurs when a local speedup leads to a larger global execution time, whereas an amplification TA occurs when a local slowdown leads to an even larger global execution time. Counter-intuitive TAs require that static timing analyses perform an exhaustive exploration of the reachable hardware states, and amplification TAs jeopardize the penalties that compositional analyses may introduce to integrate the timing behavior of some components (e.g., caches) in the total timing.

We will mainly illustrate local timing variations through cache misses and we will focus on scheduling counter-intuitive TAs in OoO pipelines and on amplification TAs caused by contention of memory accesses in in-order pipelines.

2 – MICROARCHITECTURE CASE STUDIES

We present next the various pipelined hardware microarchitectures that constitute the case studies of this work. We first introduce a textbook in-order pipeline (Sec. 2.1) that we use as a reference to study amplification timing anomalies, from which we extend previous work on this topic (in Ch. 9) to a more complex microarchitecture, namely the in-order superscalar TriCore microarchitecture (Sec. 2.2). TriCore is an industrial processor mainly used in the automotive field. It is based on an in-order superscalar microarchitecture, with advanced features (such as a Store Buffer) that make it more sophisticated that the previously studied microarchitectures and that hinder predictability. In this chapter, we present the features of its pipelines that may have an impact on amplification TAs. Then, we describe the template of a parameterizable out-of-order pipeline (Sec. 2.3) that we adopt to study counter-intuitive timing anomalies (in Ch. 5). This template is a generic, representative microarchitecture, on which the commonly represented TA pattern can manifest. We provide execution examples for each case study and we introduce the typical situations of TAs on which we based our work.

2.1 . Classical In-Order Pipeline

We consider a textbook microprocessor [9] comprised of a 5-stage in-order single-issue pipeline, a cache memory, and connected to a memory system. Hahn et al. [19] have shown that amplification TAs may occur due to contention of memory accesses even in such a simple in-order pipeline, and Jan et al. [26] analyzed various predictable pipelines with regard to such TAs.

First, we describe the basic pipeline and we exemplify an execution of a program on this pipeline (Sec. 2.1.1). Then, we introduce our guideline example of amplification TAs in in-order pipelines (Sec. 2.1.2).

2.1.1 . Description of the Pipeline

The microprocessor is connected to an external (main) memory through a memory bus. The pipeline stages form a (total) order, with the successive stages shown in Fig. 2.1:

1. Instruction Fetch (IF): This stage is responsible for retrieving the instructions of the executed program and initiating the traversal of the instructions through the pipeline stages. The instructions are retrieved either from the main memory (i.e., cache miss), or directly from the cache if they are already present inside (i.e., cache hit).

2. Instruction Decode (ID): This stage uses a combinatorial logic to decode the instructions, namely to get the opcode (instruction type) and the operands.

3. Execute (EX): This stage uses the ALU to execute the arithmetic instructions strictly speaking, i.e., making the computations on the operands.

4. Memory (MEM): This stage handles the memory instructions, i.e., data reads/writes from/to the main memory (cache miss) or the cache (cache hit). We do not worry whether the data cache is separate from the instruction cache (related to IF), but we will qualify as *data-cache misses* the misses related to MEM, in contrast to *instruction-cache misses* related to IF.[1]

5. Write-Back (WB): This stage writes the results of arithmetic operations or the data read from cache or memory to the appropriate register in the processor register file. The effective processor state, representative of the evolution specified by the ISA (cf. Sec. 1.1.1), is thus updated here.

The only two stages that exhibit a *variable timing behavior* are thus IF and MEM, due to a performance enhancer, specifically the cache. Even if we consider separate instruction and data caches, the memory bus is shared and the information (instruction or data) is multiplexed. As a consequence, **the IF and MEM stages may interfere**.

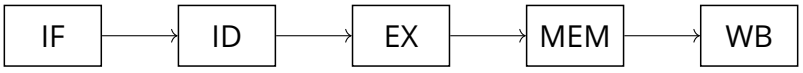


Figure 2.1: Textbook 5-stage in-order pipeline.

Let us consider a 5-instruction program made of the successive instructions A to E. Table 2.1 represents the ideal execution of this program on the pipeline, i.e., only with cache hits and no stall cycles. Table 2.1a represents a picture of the system resources (i.e., pipeline stages) in function of time, showing in any cycle which instructions are currently processed by each resource. It may be more convenient to represent pipelined executions from a second, program viewpoint, namely detailed traces showing the processing of the successive instructions in function of time. Hence, Table 2.1b represents, in an equivalent way, the trajectory of any instruction through the successive pipeline stages, in function of time. By considering the instructions as particles and the execution as a flow crossing the stages, the first viewpoint is similar to the Eulerian description in field theory, whereas the second one is similar to the Langrangian description. We can observe, in particular in Table 2.1a, that from cycle 5, one instruction completes (in WB) in each cycle, allowing the ideal throughput of 1 instruction per cycle after the transitional regime

[1]This terminology will also be used for equivalent notions in the microarchitectures introduced below.

Table 2.1: Ideal execution of a 5-instruction program on the in-order pipeline represented in Fig. 2.1.

(a) Resource, "Eulerian" viewpoint.

Stages \ Cycles	1	2	3	4	5	6	7	8	9
IF	A	B	C	D	E				
ID		A	B	C	D	E			
EX			A	B	C	D	E		
MEM				A	B	C	D	E	
WB					A	B	C	D	E

(b) Instruction, "Lagrangian" viewpoint.

Instr. \ Cycles	1	2	3	4	5	6	7	8	9
A	IF	ID	EX	MEM	WB				
B		IF	ID	EX	MEM	WB			
C			IF	ID	EX	MEM	WB		
D				IF	ID	EX	MEM	WB	
E					IF	ID	EX	MEM	WB

where the pipeline is being filled (cf. Sec. 1.2.2). In the remainder, we will focus on the second representation for any pipeline, since timing anomalies are intuitively defined from variations in the trajectory of some instructions.

2.1.2 . Guideline Example of Amplification TAs

Table 2.2 illustrates an amplification TA in the pipeline described above by comparing two traces. The anomaly is caused by memory accesses that are either serviced by a cache or require a bus access. At the end of cycle t_3, the load/store instruction A aims to perform a data memory access in the next stage (MEM), while instruction B is fetched at the same time in the IF pipeline stage. Instruction B always suffers from a cache miss when fetched ($3{\times}\mathrm{IF}$) and thus always accesses the bus. Instruction A, on the other hand, may either suffer a cache miss ($3{\times}\mathrm{MEM}$ in the trace at the top) or experience a cache hit ($1{\times}\mathrm{MEM}$, trace at the bottom). The instruction does not access the bus in the later case. The anomaly is caused by the two pipeline stages that interfere (cf. Sec. 2.1.1) and the additional *stalling* due to the bus conflict between A and B: in addition to the two stall cycles for A's cache miss ($\Delta_L = 3 - 1 = 2$), the pipeline is stalled an additional cycle at time instant t_4 due to the bus conflict ($\Delta_G = t_8 - t_5 = 3$). Table 2.2 illustrates this case as in Fig. 1.2, where the local variation $\Delta_L = 3-1 = 2$ is caused by the switch from a data cache hit to a cache miss for instruction A, while the global variation $\Delta_G = t_8 - t_5 = 3$ for the end of instruction A is greater

than Δ_L because of the instruction cache miss of instruction B.

Table 2.2: Example of an amplification TA due to fetch and data memory accesses, constituted by two different execution traces. Instruction A either suffers a data cache miss ($3\times$**MEM** in the top trace) or experiences a data cache hit ($1\times$*MEM* in the bottom trace). Instruction B always shows an instruction cache miss (**IF**).

Cycle	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
						$\Delta_L = 2$			
Instr. A	IF	ID	EX	EX	**MEM**	**MEM**	**MEM**	WB	
Instr. B		**IF**	**IF**	**IF**	ID	EX	EX	MEM	WB

Instr. A	IF	ID	EX	*MEM*	WB				
						$\Delta_G = 3$			
Instr. B		**IF**	**IF**	**IF**	ID	EX	MEM	WB	

2.2 . Overview of the TriCore Microarchitecture

This section details our more sophisticated case study for the analysis of amplification TAs: the in-order superscalar TriCore microarchitecture. This microarchitecture is sufficiently simple to be commonly embedded in real-time systems, but much more complex than the predictable pipelines that Jan et al. [26] study regarding the amplification TA introduced in Sec. 2.1.2. In Ch. 9, we show how this analysis can be extended to the complex case of TriCore.

In the remainder, we first present an overview of the TriCore microarchitecture (Sec. 2.2.1). Then, we detail its timing behavior wrt. the progression of instructions in the pipeline, the interactions between instructions, and their interactions with the memory hierarchy (Sec. 2.2.2). The timing behavior is essential in our formal modeling of TriCore, introduced in Ch. 9.

2.2.1 . The TriCore Microarchitecture

The TriCore microarchitecture is composed of two principal pipelines—an Integer (I) and a Load/Store (LS) pipeline—and a third one, specialized for hardware loops. The three pipelines form the execution unit, as described in detail in the manual [32, p.218]. In the following, we focus our investigation of amplification timing anomalies on a compact TriCore microarchitecture consisting of the two principal pipelines, as shown in Fig. 2.2. Structurally, the two pipelines are quite similar and close to the text-book in-order pipeline presented in Sec. 2.1; however, they provide different functionalities. The I-pipeline mainly handles arithmetic instructions, whereas the LS-pipeline handles load and store instructions.

The *Instruction Fetch* (IF) stages of the two pipelines operate either in-sync on so-called *fetch bundles* (i.e., pairs of I- and LS-instructions) or on a single I- or LS-instruction (meanwhile a bubble is inserted in the other pipeline) [33]. Instruction Fetch is responsible for fetching instructions from the memory system

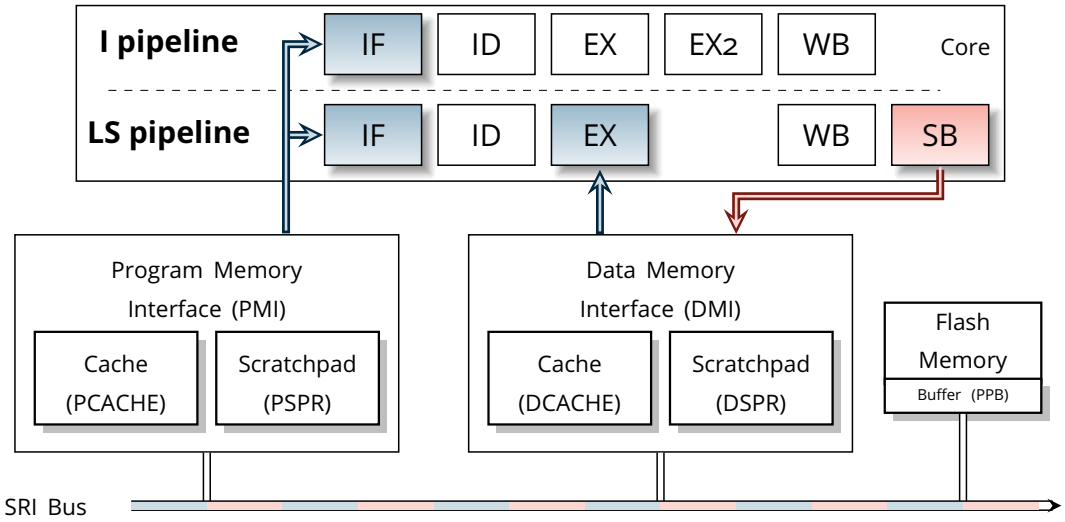


Figure 2.2: The TriCore microarchitecture: the principal pipelines and the memory system with input and output buses.

and to direct them to the appropriate execution pipeline. It is the de facto first stage of both pipelines. Both IF stages are connected to the *Program Memory Interface* (PMI) [32, p.297]. The PMI is comprised of a Program Cache (PCACHE) and a Program Scratchpad (PSPR) and is in-turn connected to the *Shared Resource Interconnect Bus* (SRI) [32, p.57]. The SRI bus allows instructions to be fetched from a Flash Memory, which is equipped with a *Program Prefetch Buffer* (PPB), and may cause the pipeline to stall.

Data memory accesses are performed exclusively by the LS-pipeline. Load instructions perform data reads in the EX stage, while store instructions place their data writes in an asynchronously operating store buffer. These components are connected to the *Data Memory Interface* (DMI), comprised of a Data Cache (DCACHE) and a Data Scratchpad (DSPR) [32, p.310]. The load and store instructions also access the SRI through the DMI (in order to communicate with the Flash Memory) and this shared resource may cause pipeline stalling. The load instructions, the store buffer, and the PMI contend for the shared SRI bus and may thus interfere with each other. Notably, any of the three components may delay an access of any of the other components. However, according to the manual [32, p.228], the store buffer has the lowest priority of all.

2.2.2 . Timing Behavior

Apart from the IF stage and the EX stage, which are connected to the memory interfaces, the other pipeline stages cannot cause stalling and complete in a single cycle. Note that only a few multi-cycle instructions (e.g., multiply-accumulate) make use of the EX2 stage in the I-pipeline. However, **the two pipelines may interact with each other—even for instructions that do not belong to the**

same fetch bundle. Data dependencies and structural hazards (cf. Sec. 1.2.2) in the I-pipeline may stall the LS-pipeline for several cycles, whereas stalls in the LS-pipeline also stall the I-pipeline for the same number of cycles [33].

Memory Latencies

Table 2.3: Baseline latencies (in clock cycles) for memory accesses in the TriCore microarchitecture.

(a) Program Memory Interface (PMI)

PCACHE hit/local PSPR	1
Flash memory access & PPB hit	4
Flash memory access & PPB miss	8
Distant DSPR (via SRI bus)	5

(b) Data Memory Interface (DMI)

DCACHE hit (1 line)/local DSPR	1
DCACHE hit (2 lines)	2
Flash memory access (DCACHE miss)	10
Distant PSPR (via SRI bus)	5

The baseline latencies for instruction fetch accesses (IF) and data memory accesses (EX/store buffer) are reported in Table 2.3. The latency values are derived from plausible configurations [32, p.172, 828]. The PMI component (Table 2.3a) gets these instructions from either the Flash Memory or the PCACHE or the PSPR (a fast component dedicated to critical code sequences). If the instruction is stored in the main memory (not in a PSPR), the instruction may be cached in the PCACHE. In case of a cache hit, the instruction can be dispatched to the IF unit immediately. In case of a cache miss, the instruction is fetched from the Flash Memory, via the SRI bus, updating the corresponding cache line. Both cases are possible: if the instruction has been buffered (in the PPB), the transfer requires 4 cycles, otherwise, an initial PFlash (Program Flash) access is necessary, incurring a penalty of 4 more cycles. An instruction can also be fetched from a DSPR, which entails a transfer via the SRI bus [32, pp.172, 297, 310].

The baseline latencies related to the DMI are reported in Table 2.3b. A hit in the DCACHE may occur at the end of a cache line and therefore span over two cache lines, entailing one additional wait cycle [32, p.310]. A DCACHE miss requires a cache line refill from the main memory and then, an initial DFlash (Data Flash) access is performed. Finally, data can also be contained in the local DSPR or a (distant) PSPR.

Pipeline Hazards

Dependent-Load Hazards. A store instruction, when followed by a dependent load instruction (i.e., that accesses the same memory address), causes a *memory*

reference hazard. In this scenario, the microarchitecture ensures that the load instruction is correctly updated by stalling it until the completion of the store instruction [33]. Table 2.4 exemplifies this situation: case (a) shows the incorrect execution that would occur if the hazards were ignored, while case (b) shows the stalling introduced to resolve it. In Table 2.4a, the value read in the EX stage is not up to date, since the store has not already written the data in the SB stage. On the contrary, in Table 2.4b, the value read in cycle t_7 is correct, due to the stalling in the EX stage.

Table 2.4: A memory-reference hazard between a store and a dependent load.

(a) Illegal execution, if a memory-reference hazard were ignored: the load is supposed to read the value of the preceding store.

Cycle	t_1	t_2	t_3	t_4	t_5	t_6
Store	IF	ID	EX	WB	SB	SB
Load		IF	ID	EX	WB	

(b) Stall cycles are introduced to resolve the hazard.

Cycle	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
Store	IF	ID	EX	WB	SB	SB		
Load		IF	ID	EX	EX	EX	EX	WB

RAR/WAR/RAW Dependencies. Besides the dependencies between memory accesses, two instructions may also present data dependencies due to their registers, as source and, respectively, destination. These dependencies could cause additional pipeline hazards and hence, require special features to guarantee the correct execution of the program. *Read-after-Read* dependencies (RAR), occur when two instructions share a source register. This does not cause any hazard, since the shared resource is read-only. Besides, *Write-after-Read* dependencies (WAR) occur when the destination register of the second instruction is a source register of the first one. Hazards associated with WAR dependencies cannot occur in the TriCore microarchitecture, since instructions always execute in-order and registers are read early in the pipeline (in the ID stage) and written late (in the WB stage). Finally, *Read-after-Write* dependencies (RAW) occur when the destination register of the first instruction is a source register of the second one. The associated hazards are resolved by forwarding [33] in the TriCore microarchitecture and thus do not cause additional delays, as in many modern pipelines.

WAW Dependencies. *Write-after-Write* dependencies (WAW), occur when two instructions share the same destination register. Contrary to the other forms of dependencies, the associated hazards may cause additional stall cycles when both instructions execute in parallel in the two pipelines. This may lead to a data

hazard (i.e., an incorrect order of writes) or a structural hazard, i.e., a resource contention at the write port of the register file. For such WAW hazards to occur, a load instruction has to execute on the LS pipeline that has a WAW dependency with either a preceding multi-cycle instruction or any arbitrary instruction in the I-pipeline [33]. In all cases, these hazards are resolved by stalling the pipeline for 1 or 2 additional cycles.

Table 2.5: Write-after-Write (WAW) hazards between a load and an I-pipeline instruction.

(a) Illegal execution, if a data hazards were ignored: instruction A is supposed to write its result before B.

Cycle	Instr.	t_1	t_2	t_3	t_4	t_5
I pipeline	A	IF	ID	EX	EX2	WB
LS pipeline	B	IF	ID	EX	WB	

(b) Two stall cycles are introduced to resolve the data hazard of (a).

Cycle	Instr.	t_1	t_2	t_3	t_4	t_5	t_6
I pipeline	A	IF	ID	EX	EX2	WB	
LS pipeline	B	IF	ID	ID	ID	EX	WB

(c) Illegal execution, if a structural hazard were ignored: both instructions would try to write to the same register at the same time.

Cycle	Instr.	t_1	t_2	t_3	t_4	t_5
I pipeline	A	IF	ID	EX	EX2	WB
LS pipeline	B_0	IF	ID	EX	WB	
	B		IF	ID	EX	WB

(d) In this case, one single stall cycles is required to resolve the structural hazard of (c).

Cycle	Instr.	t_1	t_2	t_3	t_4	t_5	t_6
I pipeline	A	IF	ID	EX	EX2	WB	
LS pipeline	B_0	IF	ID	EX	WB		
	B		IF	ID	ID	EX	WB

Tables 2.5a and 2.5b show a data hazard within a fetch bundle consisting of a multiply-accumulate (A) and a load instruction (B) that both write to the same register. Instruction A takes two cycles to execute, i.e., spends an extra cycle in the EX2 stage and thus would write its result *after* B. Table 2.5a shows the incorrect execution that would occur if the data hazard were ignored (cf. the order of the colored WB stages). In order to enforce in-order completion, instruction B in the LS-pipeline needs to stall for two cycles (in the ID stage), as presented in Table 2.5b. These stalls (are intended to) alter the relative position between instructions. The number of stall cycles vary if the I- and LS-instructions are not part of the same fetch bundle.

Tables 2.5c and 2.5d show a structural hazard due to the same interaction scenario between instructions A and B, while considering an intermediate instruction B_0 in the same fetch bundle as A. Without special handling, in the incorrect execution shown in Table 2.5c, both instructions try to write the register at the *same time*, which leads to a structural hazard. Here, Table 2.5d shows that instruction B requires a single stall cycle (in the ID stage) to solve the hazard.

2.3 . A Representative Out-of-Order-Pipeline Template

In this section, we introduce the template of an Out-of-Order (OoO) pipeline (see Sec. 1.2.2). The template is representative of modern OoO pipelines that are susceptible to TAs, for instance the RISCV BOOM core.[2] OoO microarchitectures are well known to exhibit counter-intuitive TAs [21, 22, 23, 24, 34], more documented and typically illustrated in this situation, as well as amplification TAs [21, 23]. We use this hardware template to assess the existing formal definitions of counter-intuitive TA s, in Part II.

We first provide an overview of the template (Sec. 2.3.1), before elaborating on the execution functioning (Sec. 2.3.2). Then, we introduce the traditional pattern of counter-intuitive TAs found in the literature, caused by the scheduling in OoO pipelines (Sec. 2.3.3).

2.3.1 . Pipeline Overview

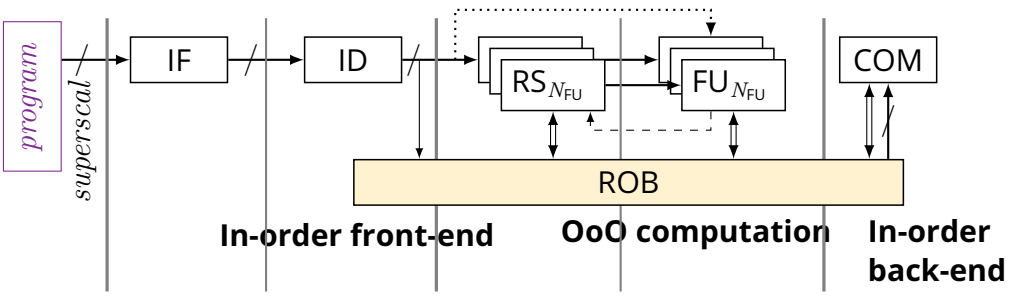


Figure 2.3: Representative hardware template of an OoO pipeline based on Tomasulo's algorithm. The pipeline has N_{FU} functional units and is able to fetch, decode and commit $superscal$ instructions per cycle from a software specification ($program$).

The template, illustrated in Fig. 2.3, is inspired by case studies proposed in the literature to reason about timing modeling [21, 35]. It is based on a pipeline containing:

[2] https://boom-core.org/

- an in-order front-end, responsible for fetching (IF) and decoding (ID) the instructions from the input sequence;
- an **OoO execution engine** in the middle, which may hold instructions in Reservation Stations (RS) and perform computations in Functional Units (FU)—while respecting (data) dependencies;
- and an in-order back-end, so as to commit (COM) the instructions in program order via a reorder buffer (ROB).

The template can be parameterized using four parameters: $superscal$ determines the number of instructions fetched/decoded/committed per cycle, N_{FU} specifies the number of RSs/FUs, and S_{RS} and S_{ROB} specify the sizes of the buffers of the RSs and the ROB.

Concrete implementations may bring out some additional stages, however only prolonging the in-order front-end or back-end, without affecting the scheduling algorithm. The template abstracts away the Write-Back (WB) stage (if existing), since, on the one hand, we assume full bypassing allowing back-to-back operations even in case of (Read-after-Write) data dependencies, and, on the other hand, we do not represent the register file explicitly (cf. Sec. 5.1.2).[3] Besides, we assume that *each* FU has a Common Data Bus (CDB) in charge of broadcasting the produced data towards the ROB and, by bypassing, towards the RSs. Hence, the number of simultaneous computation completions in the FUs is not limited.

2.3.2 . Execution Functioning

The template allows executing an arbitrary specified instruction sequence, as $program$ in Fig. 2.3. We can specify for each instruction its data dependencies and the admissible FUs. Data-cache misses/hits will be modeled through regular FUs (for loads/stores)[4] that have a variable timing behavior (cf. Ch. 5). We specify for each instruction the sets of possible latencies for the FUs and the IF stage (the COM stage always takes 1 cycle), representing for instance the behavior of, respectively, the instruction and the data caches. The instruction sequence, the choice of FUs, as well as the choice of latencies explicitly represent the *initial state* that *determines* the outcome of an execution trace. The set of all initial states is given by all possible combinations of these choices. A subset of all these initial states thus yields the set of execution traces that can actually be observed (and thus need to be analyzed for timing anomalies).

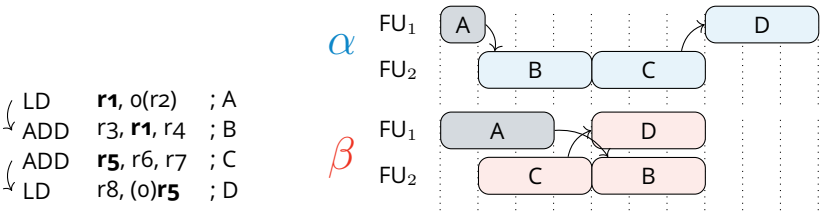
Starting from an initial state, instructions deterministically advance through the pipeline at each cycle ($\rightarrowtail$). The OoO computation relies on Tomasulo's algorithm [38] and only represents how instructions progress through the pipeline, i.e., the instruction computations will not be modeled. When several FUs are

[3] For instance, BOOM implements a Bypass Network for forwarding the Write-Back data to the register-read stage [36].

[4] Several load/store units may exist; or separate load and store units, e.g., the Pentium 4 [37].

admissible for an instruction, an arbitrary choice is made. Instructions are issued directly from the ID stage to a FU (⋯➤) or otherwise from the associated RS, when the respective FU is occupied or the instruction's data dependencies are not satisfied. The results from FUs are bypassed/forwarded (- -➤), allowing the back-to-back execution of dependent instructions on FUs, instead of delaying it until the update of the ROB. Note that in modern OoO architectures, register renaming is extensively used to fully avoid Write-after-Read (WAR) and Write-after-Write (WAW) hazards (see Sec. 2.2.2) caused by data dependencies in the program. Only Read-after-Write (RAW) hazards may still subsist in executions, despite the use of forwarding. The RSs and the ROB keep track of the status of instructions (pending/ready/executing/completed/committed) and their (data) dependencies (⟷). The oldest instruction in a RS is selected to execute on a FU among the ready instructions. Instructions are assigned an entry in the relevant RS and in the ROB within the ID stage. If the buffer capacity of one of these resources is reached (S_{RS} or S_{ROB}), the pipeline is stalled in ID (otherwise this stage takes 1 cycle).

2.3.3 . Traditional Pattern of Counter-Intuitive TAs



(a) Input assembly code. **(b)** Scheduling on functional units (FUs).

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

(c) Execution traces on our template represented in Fig. 2.3, with $superscal = 2$ and $N_{\text{FU}} = 2$.

Figure 2.4: Traditional scheduling pattern [21, 22, 23, 24, 34] (2.4b, 2.4c) on the functional units of an OoO pipeline, showing a counter-intuitive TA from a given program (e.g., 2.4a) with data dependencies (⇂).

Fig. 2.4 introduces a common example of a counter-intuitive TA [21, 22, 23, 24, 34], caused by a variation in the number of cycles that instruction A spends in functional unit FU_1, which impacts the instruction scheduler of an OoO processor

executing a sequence of instructions such as that of Fig. 2.4a with (read-after-write) data dependencies. The variation in FU_1 may represent a cache hit in case α, against a cache miss in β. Fig. 2.4b shows that the favorable variation, i.e., the local speedup in α, leads to an actual in-order scheduling on the FUs that entails a larger global execution time than in β; contrariwise, instruction C can execute actually out of order and earlier in case β, which in turn allows dependent instruction D to execute—and thus the global execution to complete—earlier. This is a typical situation of a counter-intuitive TA. Fig. 2.4c shows that this counter-intuitive-TA pattern can be obtained exactly from our template introduced in Sec. 2.3.1, for instance with $superscal = 2$ and $N_{\mathsf{FU}} = 2$. This figure represents cases α and β as two complete, *detailed execution traces* of an input $program$ representing the instruction sequence shown in Fig. 2.4a with its data dependencies, on our OoO-pipeline template, according to the behavior described in Sec. 2.3.2.

3 – FORMAL VERIFICATION

In this chapter, we provide the background on the formal verification of real-time systems, from general notions to the tools used in next parts.

In order to reason about a system, one needs a *model* of the system. The operation permitting to pass from a real system (such as a real-time system) to a model of this system is *abstraction*. An abstraction can be considered as a generalization that consists in describing the system by focusing on some particular aspects of interest only, necessarily with approximations for other aspects. Obviously, several models may exist, depending on the type of targeted aspects and the level of granularity. A given abstraction may be *refined* by adding details to an (abstract) model in order to get a more concrete model, i.e., closer to the real system.

Specifying a system—namely, listing the (functional and non-functional) requirements that the system must satisfy—can be achieved at various levels of detail, and of confidence furthermore. The intended system at the design stage is necessarily an abstract model of the final, real system. We will often make an equivalent use of the words *specification* and *model*, without suggesting any stage of the lifecyle of the system. That means in particular that a specification may describe the actual behavior of the (real) system, as well as of the expected one. When a specification of a system is developed, a certain part of its behavior can be described confidently, since the real system has been observed or will be surely conforming. However, some requirements may be expressed as constraints that must be satisfied, depending on other requirements, thus not fulfilled a priori. Besides, the real system may have a behavior that was not expected or considered in the design stage. We consider here that the system specification/model itself concerns the part of the description known confidently, to which we add a set of *properties* specifying requirements that are to be checked.

In order to specify the system with a high level of confidence, i.e., to describe it unambiguously and to reason about it rigorously, *formal* models are required. Formal models use mathematical theories to describe the system. Then, it is thus possible to formally verify properties, expressed in these theories, that the system must satisfy. Hereafter, we introduce general formal notions (Sec. 3.1), before focusing on model checking (Sec. 3.2), a particular formal-verification method. Then, we introduce the modeling and verification tools (Sec. 3.3) that we use in our work. Finally, we provide a brief historical overview of applications of formal verification (Sec. 3.4).

3.1 . Formal Notions

Systems can be described using a set of (state) *variables*.

Definition 3.1: State — A *state* of a system is an assignment of a value to each state variable, from the domains of the state variables. The set of all states is denoted as $\mathcal{S}$ and is called the *state space*.

We consider discrete-time systems, so that their behavior can be represented as a sequence of states. A transition (or step) is a pair of successive states allowed by the system behavior. A common way of modeling systems is the concept of *transition system*.

Definition 3.2: Transition System (TS) — A *Transition System* is a tuple $(\mathcal{S}, \mathcal{I}, \rightarrow)$, where $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the *transition relation* (or *next-state relation*).

A transition system formally describes all possible behaviors, namely how the system may pass from a state to another. The initial states are those from which one may start observing the system, and relation $\rightarrow$ exactly contains all possible transitions.

Definition 3.3: Execution of a TS — An *execution* (also called *run* or *behavior*) of the TS is a (potentially infinite) sequence of states $(s_1, s_2, \ldots) \in \mathcal{S} \times \mathcal{S} \times \ldots$ s.t. (such that) the successive states are in the transition relation: $\forall i \in \mathbb{N}^*$, $(s_i, s_{i+1}) \in \ \rightarrow$. It is an *initial* execution if $s_1 \in \mathcal{I}$.

The *diameter* of a TS is the least number of steps to reach all reachable states (i.e., the states that belong to at least one initial execution).

A convenient way to characterize a transition from a given, current state lies on *actions*. An action is a formula that relates to two states. Let us consider a pair of two states $(S_1, S_2) \in \mathcal{S} \times \mathcal{S}$, where S_1 is called *current (old) state* and S_2 is called *next (new) state*. The (unprimed) variable symbol x thus refers to the (current) value of variable x in state S_1, whereas x' refers to the (next) value of the same variable in state S_2.

Definition 3.4: Action — An *action* is a formula mixing unprimed variables with primed variables (thus the current values of one or several variables with their next values).

Any transition $(s_1, s_2) \in \ \rightarrow$ is formed by two successive states $s_1 = S_1$ and $s_2 = S_2$ of a possible execution of the TS. From the above (with $s_1 = S_1$ and $s_2 = S_2$), any transition can be labelled by an action (possibly involving all state variables and their current and next values). A transition is thus characterized by the current state (s_1) and the action label.

Definition 3.5: Labelled Transition System (LTS) — A *Labelled Transition System* is a tuple $(\mathcal{S}, \mathcal{I}, \Lambda, \rightarrow_\Lambda)$, where $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, Λ is a set of *labels*, and the transition relation $\rightarrow_\Lambda$ is a set of labelled transitions.

Let us assume that action $a \in \Lambda$ relates to states s_1 and $s_2 \in \mathcal{S}$. We can

construct an LTS with actions as labels, s.t. if a is true, then a is true *of the transition* $(s_1, s_2) \in \rightarrow$, namely there exists a TS execution where s_1 and s_2 are successive states. In this case, a, s_1, and s_2 form a labelled transition of the LTS, i.e., a is *enabled* at state s_1 and $(s_1, a, s_2) \in \rightarrow_\Lambda$. Conversely, the transition relation $\rightarrow_\Lambda$ of the LTS is a subset of $\mathcal{S} \times \Lambda \times \mathcal{S}$.

Executions of an LTS are of the form $(s_1, a_1, s_2, a_2, s_3, \ldots)$, where the successive labelled transitions are in the transition relation: $\forall i \in \mathbb{N}^*$, $(s_i, a_i, s_{i+1}) \in \rightarrow_\Lambda$. However, we can define *paths* of an LTS from executions, by removing all labels/actions. In short, the path corresponding to the above execution is $(s_1, s_2, s_3, \ldots)$.

Now, suppose that one intends to assess a property over a system state.

Definition 3.6: State Function/Predicate — A (state) function is a formula that can contain (state) variables, as well as constants. A Boolean-valued (state) function is called a (state) predicate.

Many properties targeting a single state of the system can be expressed as first-order logic formulae, to which we will restrict (with background theories, such as arithmetic). First-order logic formulae are inductively built from rules involving *symbols*.

Definition 3.7: Syntax of a First-Order Formula — The symbols of a first-order logic formula are:

- logical connectives (as in propositional logic) and punctuation symbols (e.g., parentheses);
- an equality symbol ($=$);
- quantifiers ($\forall$ for universal quantification and $\exists$ for existential quantification);
- variables ranging over individual objects (denoted by letters);
- truth constants ($\top$ for *true* and $\bot$ for *false*);
- function symbols (cf. Def. 3.6;
- predicate (or relation) symbols (cf. Def. 3.6.

Constant symbols refer to 0-ary functions.

The syntax of first-order formulae relies on logical symbols and non-logical symbols. Whereas logical symbols (e.g., the equality symbol $=$ and logical connectives) have a fixed semantics and are interpreted a priori, non-logical symbols (i.e., function/constant and predicate symbols) require an interpretation in order to give the formulae a truth value ($\top$ or $\bot$).

Definition 3.8: Interpretation for a First-Order Formula — An interpretation $\mathcal{L}$ of an arbitrary first-order logic formula φ is comprised of the following elements:

- a domain of discourse $\mathcal{D}$;

- an assignment of an element of $\mathcal{D}$ to every constant symbol;
- the interpretation of every n-ary uninterpreted[a]-function symbol, i.e., a function $\mathcal{D}^n \to \mathcal{D}$;
- the interpretation of every n-ary uninterpreted-predicate symbol, i.e., the subset of $\mathcal{D}^n$ that makes the predicate true.

The truth value of formula φ under interpretation $\mathcal{L}$ is denoted as $[\![\varphi]\!]_{\mathcal{L}}$.

[a] In view of the considered background theories.

Assigning a meaning to all sentences of first-order logic requires a so-defined interpretation. An arbitrary (well formed) first-order formula may be true or false, depending on the considered interpretation.

Definition 3.9: Model for a Formula — Given a formula ϕ, an interpretation $\mathcal{L}$ that satisfies ϕ, i.e., $[\![\varphi]\!]_{\mathcal{L}} = \top$, also denoted as $\mathcal{L} \models \varphi$, is called a model for formula φ.

If $\mathcal{L}$ is not a model for φ, we denote it by $\mathcal{L} \not\models \varphi$.

Definition 3.10: Satisfiability of a Formula — Formula φ is:

- *satisfiable* if there exists an interpretation $\mathcal{L}$ that is model for φ, i.e.:
$$\exists \mathcal{L},\ \mathcal{L} \models \varphi$$
- *unsatisfiable* if there exists no interpretation that is model for φ, i.e.:
$$\forall \mathcal{L},\ \mathcal{L} \not\models \varphi$$
- *valid* if any interpretation is model for φ, i.e.,
$$\forall \mathcal{L},\ \mathcal{L} \models \varphi, \text{ also denoted as: } \models \varphi$$

.

If we consider a system modeled by an LTS, we need to enrich the modeling structure with an interpretation to evaluate formulae on states. However, it may be more convenient to proceed in an indirect way. In this context, predicates are state predicates; they depend on the state variables.

Definition 3.11: Kripke Structure (KS) — Let AP be a set of atomic predicates. A *Kripke Structure* is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \Lambda, \to_\Lambda, AP, \mathcal{L})$ that is an LTS enriched with a *state-labeling* (or *interpretation*) function $\mathcal{L} : \mathcal{S} \to \mathcal{P}(AP)$ (where $\mathcal{P}$ designates the power set), providing for every state the set of all atomic predicates in AP that are satisfied in the state.

The state-labeling function provides the minimal information in order to determine whether any formula derived from the atomic predicates is true or not of a state.

Paths of a KS are similar to paths of an LTS. If we focus on the sequence of (true) atomic predicates instead, we can define traces. The trace of path $(s_1, s_2, s_3, \ldots)$ is $(\mathcal{L}(s_1), \mathcal{L}(s_2), \mathcal{L}(s_3), \ldots)$. The set of all traces allowed by the

KS is denoted as $\mathcal{L}(\mathcal{M})$.

In the next chapters, we will often represent sequences of states only. We will use the term *trace* with a more general meaning than the strict one used in the current chapter for introducing the formal notions, potentially also referring to executions of TSs and paths of LTSs. We will also use the phrase *(execution) trace* to describe sequences of states representing one execution of a program on a microarchitecture model.

3.2 . Model Checking

The main advantage of writing *formal* specifications of systems is that we can verify properties of these specifications: this is *formal verification*. Formal verification can be (highly) automated, by means of computer tools. Among the automated-proof techniques, we can cite (automatic) theorem proving [39] and model checking [40, 41, 42, 43]. Theorem proving is deductive reasoning on a system description, through step-by-step inference rules. Model checking relies on a model checker to perform an exhaustive search of a property violation, over all possible states allowed by the specification. Theorem proving may turn to be tedious and to require extensive manual effort to prove a property. Contrariwise, model checking needs fewer manual efforts, since many properties can be expressed directly from the formal specification. Model checking has consequently gained notoriety in industry, and many tools have been developed [44, 45, 46, 47, 48, 49]. However, model checking faces the state-explosion problem, due to its exhaustive nature. For this reason, a considerable effort may have to be made to adopt a suited abstraction level while writing the formal specification. **In this thesis, we focus on model checking to verify properties related to timing anomalies.**

3.2.1 . Invariants

Since model checking is based on state exploration and property assessment over states or traces, the Kripke structure (Def. 3.11) proves to be one of the natural formal structures that can be manipulated by a model checker. So, we consider model $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \Lambda, \rightarrow_\Lambda, AP, \mathcal{L})$. We focus on first-order logic to express state properties, for which models are simple interpretations as defined in Def. 3.8. However, we are generally interested in verifying properties dealing with several states that represent the system at different instants; we need modal, *temporal logics* to write formulae that express the properties easily (without stating time explicitly) [50, 51]. We will exclusively verify **linear-time properties**, which apply to *traces* of the Kripke structure.

Definition 3.12: Linear-Time (LT) Property — A linear-time property over AP is a subset of $(\mathcal{P}(AP))^\omega$, i.e., of the set of (possibly infinite) sequences whose elements are in $\mathcal{P}(AP)$.

Definition 3.13: Model for an LT Property — Trace $t \in \mathcal{L}(\mathcal{M})$ is model for a linear-time property P over AP iff (if and only if) $t \in P$, which is denoted as $\mathcal{M}, t \models P$. KS $\mathcal{M}$ is model for P iff $\mathcal{L}(\mathcal{M}) \subseteq P$, which is denoted as $\mathcal{M} \models P$.

Trace t violates property P if $t \notin P$.

To express linear-time properties, we can adopt specialized temporal logics, for instance **Linear Temporal Logic (LTL)** [41] or a variant, **Temporal Logic of Actions (TLA)** [48]. Linear-time properties are basically divided into *safety* and *liveness* properties [51]. Safety properties specify that "something bad will never happen", whereas liveness properties specify that "something good will eventually occur". **We will focus on *safety* properties, more specifically on *invariant* properties, i.e., safety properties that refer only to the current state.**

Definition 3.14: Invariant — A (linear-time) property is an invariant if there exists a state predicate φ s.t. $P = \{(A_1, A_2, A_3, \ldots) \in (\mathcal{P}(AP))^\omega \mid \forall i \in \mathbb{N}^*,\ A_i \models \varphi\}$. Such a state predicate φ is called an *invariant* (condition).

In LTL and TLA, we can express above invariant property P as follows: $P = \Box\,\varphi$, or equivalently in LTL: $P = \mathbf{G}\,\varphi$, where φ is a state predicate.

A model-checking problem for an invariant thus consists in:

1. specifying the system (e.g., through KS $\mathcal{M}$);
2. formalizing the property in a temporal logic (e.g., invariant φ);
3. deciding whether $\mathcal{M} \models \Box\,\varphi$ (through dedicated algorithms).

The last step is performed automatically by the model checker. According to Def. 3.13 and 3.14, it holds:

$$\mathcal{M} \models \Box\,\varphi \iff \forall t \in \mathcal{L}(\mathcal{M}),\ \mathcal{M}, t \models \Box\,\varphi \tag{3.1}$$

If the property is verified, the model checker proves that it holds for all traces. Otherwise, i.e., $\mathcal{M} \not\models \Box\,\varphi$, the model checker proves that there exists (at least) a path on which the property does not hold, by providing a *counterexample*, namely a trace/path that violates the property. Note that if a safety property is violated, there always exists a finite counterexample.

3.2.2 . Explicit Model Checking

According to Eq. 3.1 and Def. 3.14, a KS is model of an invariant iff every reachable state is model of the invariant:

$$\mathcal{M} \models \Box\,\varphi \iff \forall t = (\mathcal{L}(s_1), \mathcal{L}(s_2), \mathcal{L}(s_3), \ldots) \in \mathcal{L}(\mathcal{M}),\ \forall i \in \mathbb{N}^*,\ \mathcal{L}(s_i) \models \varphi \tag{3.2}$$

The KS can be unrolled into a state-transition graph that enumerates each state and each transition. The most direct manner to decide whether $\mathcal{M} \models \Box\,\varphi$ is to perform an *explicit exploration* of the state space by constructing this graph. In this way, all reachable states are visited (at least if the property is satisfied).

Then, specialized decision procedures lie on the state-transition graph to solve the problem [40].

3.2.3 . Symbolic Model Checking

One way to handle state explosion is to affect the representation of the state space, in order to reduce the size of the data that must be held in memory while model checking, instead of reducing the number of states. Symbolic model checking [52, 53, 40] allows using formulae—i.e., with *symbols*—to encode the state space, instead of explicitly building the state-transition graph.

Binary Decision Diagrams (BDDs)

Symbolic methods constituted a major step in formal verification and allowed applying model checking in industrial case studies [54]. These methods allow implicit explorations of the state space, in the sense that the state-transition graph is never built explicitly. The main symbolic methods lie on Binary Decision Diagrams (BDDs) to encode the state-transition graph in a compact way [55, 42]. BDDs [56] are representations of Boolean functions that are compact, since they operate on sets and concretely allow for factorizing shared information. The basic idea is that Boolean functions, called *characteristic functions*, can be used to represent sets and relations, thus in particular set of states and the transition relation of a KS. BDDs are used to represent the characteristic function of the transition relation, so that all paths of the KS are encoded symbolically. Distinct BDDs may be built for the same function, according to the canonical form that fixes the ordering for the (state) variables.

Bounded Model Checking (BMC)

Implicit model checking based on BDDs may still suffer from state explosion. Besides, the efficiency of their symbolic state-space representation is strongly dependent on the canonical form [42]. Bounded Model Checking (BMC) [57, 58] consists in transforming the exploration problem of model checking into the *satisfiability* problem of a formula that is satisfiable iff there exists a *finite* fixed-length sequence of transitions in the KS reaching a state that violates the property.

A Boolean Satisfiability (SAT) problem is a decision problem consisting in determining whether a Boolean formula with propositional variables is *satisfiable* or, on the contrary, *unsatisfiable* (cf. Def. 3.10).

Satisfiability Modulo Theories (SMT) is a decision problem addressing the satisfiability of a first-order logic formula (with equality and without quantifiers), wrt. associated background theories [59]; propositional variables are replaced by atomic predicates wrt. a basic SAT problem, thus interpretations also give a meaning to the predicate symbols (cf. Def. 3.8). BMC is thus generally an SMT problem, requiring an SMT solver to perform the verification. SMT uses Boolean expressions but, contrary to BDDs, they do not rely on canonical forms.

Formula F, whose satisfiability is to be determined, contains in a conjunction,

on the one hand (between square brackets below), the symbolic k-unrolling of the KS, translating all paths of length k by using k copies of the symbols that represent the state variables, and, on the other hand, the negation of the invariant φ:

$$F = \left[I(s_1) \wedge \bigwedge_{i=2}^{k} T(s_{i-1}, s_i) \right] \wedge \neg\varphi(s_k) \tag{3.3}$$

where I is the initial predicate (true iff s_1 is an initial state, i.e., $s_1 \in \mathcal{I}$) and T is the transition-relation predicate (true iff s_{i-1} and s_i form a transition, i.e., $\exists a \in \Lambda$, $(s_{i-1}, a, s_i) \in \ \rightarrow_\Lambda$). I and T are also called *characteristic functions* (see Sec. 3.2.3). F describes an under-approximation of the state space, since paths are restricted to length k. If F is satisfiable, then $\mathcal{M} \not\models \Box\varphi$ and we get a counterexample of length k. Using the equivalent contrapositive: if $\mathcal{M} \models \Box\varphi$, then F is unsatisfiable. However, the reverse is wrong: if F is unsatisfiable, then the property is true of any path *of length* k (i.e., it merely holds that $\mathcal{M} \models_k \Box\varphi$). BMC can be performed successively with increasing values of k. This permits getting counterexamples quicker, since counterexamples of minimal length are found.

3.2.4 . Counterexample-Guided Methods

The obtainment of counterexamples may provide information about the system itself (in which situations a property is violated) or about the underlying model of the system. In any case, one can exploit the obtained counterexamples to adapt the model of the system by preventing certain behaviors.

The basic use of counterexample-guided methods is *Counterexample-Guided Abstraction Refinement* (CEGAR), which aims at refining the model of a system in an automatic manner [60]. Whenever a counterexample is found, this method determines whether it represents an actual property violation or an abstraction artifact, due to a too coarse model. In the latter case, the model is refined by taking into account the infeasible behavior that must be excluded.

In Ch. 10, we set up similar strategies to explore various sources of delays in a pipeline model. We must successively exclude already explored scenarios. These strategies are based on SMT counterexample-guided approaches, with applications spanning from program synthesis [61] to microarchitecture design [62]. SMT-based counterexample-guided methods are convenient, since SMT solvers can be manipulated on the fly and integrated within an automatic specialized procedure (cf. Sec. 3.3.1).

3.3 . Modeling and Verification Tools

We intend to develop generic procedures relative to TAs. To do so, we need to model the cycle-accurate behavior of the microarchitectures that we study. We do not need to use specific time abstractions, and the procedures that we

develop are not tool-specific. We base our formal modeling on generic (labelled) transition systems, in which the states represent clock cycles, and transitions clock ticks. We have experimented with several tools for describing our pipeline models and verifying timing properties, in particular UCLID5(from an existing modeling basis), but also TLA^+.

Next, we present the tools that we use in the following parts of this thesis to verify invariants—UCLID5 and an SMT solver (Sec. 3.3.1), and TLA^+ (Sec. 3.3.2). Modeling tools and languages are useful to express symbolic representations of system models (typically a KS, Def. 3.11). These representations could theoretically be generated automatically; however, all models used in this document are written manually. Model checkers and SMT solvers use these representations to verify properties.

3.3.1 . UCLID5

UCLID5 provides a specialized modeling language [63] and has been applied, in particular, for the functional verification of processors [64]. It offers abstractions to describe computational system models and supports a range of techniques to formally verify properties on these models. In our work, we exclusively use BMC.

Infrastructure

LTL and invariant properties in particular (cf. Sec. 3.2.1) can be specified and then verified by BMC (cf. Sec. 3.2.3). In this case, model checking is performed up to a *specified* number of transitions (depth) that represents the *bound*. The LTL temporal operator $\mathbf{G}$ applied to a predicate φ, i.e., $\mathbf{G}(\varphi)$, means that φ is supposed to be an invariant—to hold in all states.

UCLID5 resembles a programming language, close to C with nevertheless two specialized blocks to describe the transition-system structure: `init` and `next`. Models are composed of a set of typed variables, whose values define states. The *next* block specifies the next value of each changed variable with parallel assignments, i.e., all changes are simultaneously applied at the end of the block. The prime operator ($'$) refers to the new value of a variable. Each execution of the *next* block corresponds to a transition. The notion of time, e.g., with clocks [65], is not inherently present in UCLID5: it has to be modeled explicitly by a state variable.

Models may also explicitly define the initial values of variables in the separate *init* block. Variables that are not explicitly initialized assume arbitrary values, in accordance with the variables' types/domains. Moreover, *assumptions* can be formulated to control at any step the range of possible values for a variable, which might also introduce non-determinism. These two points draw up the state space of the verification problem. When a property is violated at a certain depth, the model checker provides a (single) counterexample for that depth, showing a detailed trace with the evolution of specified variables up to this depth.

The canonical pipeline model [26], a particular hardware abstraction designed to track delays induced in a pipeline (cf. Sec. 4.3.2), had been encoded in UCLID5.

We extended this work to TriCore, the industrial case study introduced in Sec. 2.2. In Ch. 9, we use symbols of first-order logic (connectives and quantifiers) to write term definitions ($\triangleq$), instead of the counterpart C-like logical operators and expanded forms used for variable assignments in UCLID5. Similarly, all formulae that do not comprise a definition (symbol $\triangleq$) are in fact encompassed within an `assume()` statement, in a close form to the logical one used in this document.

Unfortunately, the verification tool does not provide any statistics. We also faced various expressiveness limitations when using complex/nested data structures. Note that we do not rely on the specific abstractions offered by the language, since we do not model the results of operations.

Bounded Model Checking (BMC)

UCLID5 transforms the BMC problem (expressed by a specialized command `bmc`), performed by unrolling the transition system up to a given depth k, into one SMT problem per depth. The verification at a given depth of a property expressed in UCLID5 relies on the SMT formulation at this depth, which is checked using the Z3 solver [66]. The user has the possibility to only generate the SMT formulations (for each property and each depth up to the specified bound), without calling the solver. The successive values of the state variables are expanded into distinct symbols and the problem boils down to one conjunction that involves the symbols related to the initial state, the transition relation applied k times, and finally, the values after the last transition representing the states whose reachability is assessed, as introduced above (Sec. 3.2.3).

UCLID5 generates SMT files, in the SMT-LIB standard language [67] (supported by the SMT solvers, like Z3). Moreover, SMT solvers provide specialized APIs to directly manipulate the SMT problem. In this work, we use the Z3 SMT solver and its Python API (details are in Ch. 10). A convenient feature provided through SMT-LIB—in particular, through the API of Z3—is *scoping*. Scopes allow for directly manipulating the SMT encoding using the `push` operation, to add new assertions on the solver stack, and the `pop` operation, to remove assertions from the same stack. A direct use of scoping is that the solver can be used to incrementally address a verification problem (as in Ch. 10 in order to get multiple counterexamples).

3.3.2 . TLA$^+$

We have also experimented with TLA$^+$ [48]. It is a formal language accompanied by a complete toolbox and was originally designed for specifying concurrent systems. A specification is made of TLA formulae (cf. Sec. 3.2.1), i.e., based on first-order logic, set theory, and actions (Def. 3.4). The toolbox comes with an explicit model checker, TLC, which makes it possible to check properties on the specification. Note that we can use a symbolic model checker as an alternative [68]. Note also that the framework comes with TLAPS [69], a Proof System allowing for writing proof obligations in the same language as the specification—

TLA^+—and to send them to various back-end verifiers (e.g., theorem provers and SMT solvers).

The language is intuitive (it relies on basic mathematics), allows modularity, and is well-documented. The toolbox offers convenient features, such as numerous options for displaying the counterexample traces or the possibility of evaluating expressions at each state of a derived trace. Moreover, the model checker provides clear errors and avoids unintended state-space explosion by prompting the user to define a next-state relation for each state variable. TLA^+ does not come with specific abstractions for hardware features (e.g., bit vectors) or time but, as mentioned above, we do not need such abstractions—we use a standard state variable to model time. **We have chosen TLA^+ (with TLC) as our main formal tool for the most recently proposed work**, due to its convenient features for developing and testing models efficiently, and the range of verification techniques that are supported and integrated together into the toolbox. In particular, we have formalized the parameterizable OoO model introduced in Sec. 2.3 in TLA^+ (cf. Sec. 5.1).

TLA^+ allows us to specify a KS through a single TLA formula, containing in particular an initial-state predicate and a next-state relation built from actions relating the values of variables in the current state (e.g., x) to their values in the next state (x'). Fig. 3.1 shows a sample of the specification of our OoO template in TLA^+. The state variables are declared after the keyword VARIABLES, here a variable representing the current cycle and the program counter. The specification defines several *operators*, i.e., well-formed, named TLA^+ expressions that may be parameterized. $TimeProgress$ is an operator used later for the next-state relation. We can make several remarks from this operator:

- An IF-THEN-ELSE construct is available to alleviate expressions.
- The LET-IN construct permits writing *local* definitions.
- The expression $currCycle' = currCycle + 1$ is an atomic action (it relates to a single state variable).
- $\langle$ and $\rangle$ define sequences (i.e., functions defined on $\mathbb{N}^*$).
  The expression: UNCHANGED $\langle currCycle \rangle$ is a syntactic sugar for stating an action where each variable of the sequence is unchanged, here merely:

  $currCycle' = currCycle$

In Fig. 3.1, the possible initial values of the state variables are those that satisfy the initial-state predicate $Init$. For the two variables that are visible, a single value (0) is allowed in the initial state. The vertical alignment of logical connectives describes precedence (implication has the lowest precedence on the same vertical axis), e.g., in the figure, each line of predicate $Init$ (lines 6-8 in Fig. 3.1) is one conjunct and all conjuncts are in the same level, so the

vertical alignment does not introduce any precedence. Operator $Next$ is the next-state relation, composed of several actions in conjunction (operator $Progress$ is not shown here). Finally, the behaviors allowed by the system are those that satisfy operator $Spec$ (which is explicitly stated in the toolbox as the temporal formula that specifies the system). This formula specifies that operator $Init$ is indeed the initial-state predicate (the expression must be true of the initial state, since it is not prefixed by any temporal operator) and that operator $Next$ is indeed the next-state relation (the action must be true of any transition, see the temporal operator $\Box$). The syntax $[Next]_{\langle currCycle,\, pc,\, \ldots\rangle}$ is a syntactic sugar for $Next \vee (currCycle' = currCycle \wedge pc' = pc \wedge \ldots)$. Thus, besides the steps allowed by operator $Next$, stuttering steps (in which the specified variables are unchanged) are also allowed. Stuttering steps are mandatory for legal temporal formulae in TLA (they notably ease the composition of specifications).

variables $currCycle,\ pc,\ exec\_inst,\ \ldots$

$$
\begin{array}{lll}
TimeProgress & \triangleq & \text{if } \exists\, i \in 1\,..\,Len(pipe\_stages) : \\
 & & \quad \text{let } x \triangleq pipe\_stages[i] \text{ in } x' \neq x \\
 & & \quad \text{then } currCycle' = currCycle + 1 \\
 & & \quad \text{else unchanged } currCycle \\
Init & \triangleq & \wedge\ currCycle = 0 \\
 & & \wedge\ pc = 0 \\
 & & \wedge\ \ldots \\
Next & \triangleq & Progress \wedge TimeProgress \wedge \text{unchanged } exec\_inst \\
Spec & \triangleq & Init \wedge \Box[Next]_{\langle currCycle,\, pc,\, exec\_inst,\, \ldots\rangle}
\end{array}
$$

Figure 3.1: Sample of our TLA$^+$ specification of the OoO template (cf. Sec. 2.3).

3.4 . Applications of Formal Verification

Model checking was early used for software verification, in particular to check communication protocols [70]. Then, significant work has been done to check C programs [71, 72, 73], in traditional or embedded-system contexts. The goal is to detect errors introduced in the code (e.g., invalid pointers) and, thus, to prevent bugs. In such an approach, the underlying hardware microarchitecture is completely out of scope and is not modeled, whereas we consider timing properties of the *execution* of a program on a hardware target.

Formal verification has also been used to check the correctness of hardware designs, since the eighties [74]. It follows from Ch. 1 that *implementations* do not materialize ISAs (cf. Sec. 1.1.1) in a straightforward way, notably due to the necessary stalling cycles (e.g., for respecting data dependencies) and the performance enhancers such as out-of-order-execution engines. Since the ISA specifies the expected functional behavior of instructions, it can be seen as an abstract model

of the processor. The formal verification of hardware designs consists in checking whether a model of the implementation, called the concrete model, produces the same results as the abstract model.

Burch and Dill [75] laid the foundation for the verification of hardware designs. They modeled the control path of in-order microarchitectures, assuming that the data path was correct, and used uninterpreted functions to represent executions symbolically. Their method results in a logical formula that is valid iff the implementation is correct. However, a special decision procedure is needed for checking the validity of the formula, which prevents this method from being directly usable with model checking. Burch [76] introduced abstractions in the specification and simplifications in the decision procedure, and he extended the previous work to superscalar pipelines. Skakkebæk et al. [77] continued in this direction, by providing a support for out-of-order pipelines. Berezin et al. [78] shifted paradigms, by verifying OoO implementations with BDD-based symbolic model checking (cf. Sec. 3.2.3) while preserving uninterpreted functions, which thus no longer requires special decision procedures. They also introduced a special data structure to share sub-terms appearing in symbolic executions; pointers to entries of this data structure allow for a compact encoding. However, some manual guidance was still inevitable. Lahiri et al. [64] proposed a more systematic approach, also with uninterpreted functions but, additionally, with lambda expressions, using UCLID (UCLID5's ancestor, cf. Sec. 3.3.1). Their work mainly relies on the decision procedure of these tools that is based on inductive invariant checking. All these approaches seek to check the *functional* correctness of hardware designs, namely to verify that the relevant part of the state of the implementation is correct wrt. to the specification. Firstly, these approaches are code-independent—they do not consider any input program but, instead, any sequence of instructions allowed by the ISA—and thus more general than required in our context, since real-time systems are designed to execute specific input programs. Secondly, these methods are too abstract to verify timing properties, since, on the one hand, the guidance specifications do not deal with the *temporal* correctness, and, on the other hand, only the results of computations are considered in the implementation.

Model checking has also been used to verify timing properties, which requires taking into account both the software and the hardware. Our work lies in this context and we elaborate on this part of the related work in Sec. 4.4.1.

3.5 . Summary: our Formal Framework

In the current chapter, we introduced formal notions and formal modeling and verification tools—UCLID5 and TLA$^+$. Whereas formal verification is often performed on hardware and software separately, and focuses mostly on functional correctness, we intend to carry out the co-verification of non-functional, timing properties on hardware and software; properties thus concern (pipelined) executions

resulting from the combination of both components. We highlighted in the previous chapters the importance of verifying the absence of timing anomalies or precisely identifying them. From this formal framework, we develop in next parts formal models of the two main case studies—the representative out-of-order pipeline and the TriCore microarchitecture—aiming at detecting and analyzing timing anomalies by model checking.

4 – RELATED WORK

In this chapter, we introduce the related work specific to timing analysis and timing anomalies in microarchitectures, and we highlight the issues and limitations in the state of the art that motivate the work described in the next parts.

Counter-intuitive TAs have a systematic impact on analyses (cf. Sec. 1.3.2), whereas the effect of amplification TAs, which are an issue for compositional analyses only, is often deemed harmless [79, 23]. The particular relationship between counter-intuitive TAs and WCET analyses questions the intuitive definition introduced in Sec. 1.3.1 and complicates the impact on the WCET. We thus first describe the various interpretations of counter-intuitive TAs that coexist in the literature (Sec. 4.1). Then, we provide an overview of the existing definitions of TAs (Sec. 4.2), with an emphasis on counter-intuitive TAs. Finally, we present some predictable cores (Sec. 4.3) and we elaborate further on the work related to amplification TAs that arise from memory interference, introduced in Sec. 2.1.2, before describing modeling approaches related to general low-level WCET analysis or to our microarchitecture case studies (Sec. 4.4).

4.1 . Interpretations of Counter-Intuitive TAs

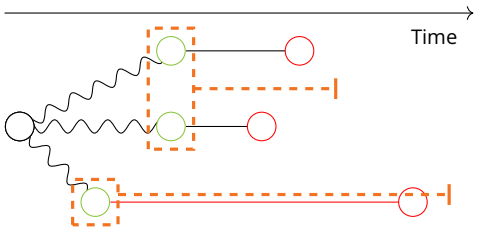
A study of the literature shows that counter-intuitive TAs can be interpreted in various ways against their impact on the WCET.[1] We have observed that the source of TAs is subject to two interpretations as to the *abstract* vs. *concrete* nature of the considered traces: respectively, a *static-analysis-centric* vs. a *hardware-centric* interpretation. Among the considered traces, those that are amenable to triggering a TA also constitute a complementary interpretation, according to whether the impact on the WCET is *absolute* or *pairwise*.

4.1.1 . Concrete and Abstract Models

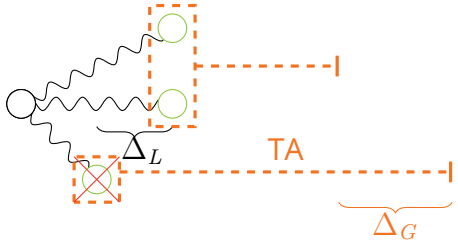
Processors are commonly modeled using transition systems, comprised of (hardware) states and transitions. Static WCET analyses compute abstract states that over-approximate the states that may appear on the concrete hardware while executing a given program, and derive timing bounds from these states (see Sec. 1.1.4). Thus, we can reason on two models: the *concrete* model serves as a reference and is assumed to represent the real system, faithfully and accurately wrt. its timing behavior; the *abstract* model provides timing estimates against a certain analysis method.

Fig. 4.1a represents a concrete model (solid shapes) and a corresponding ab-

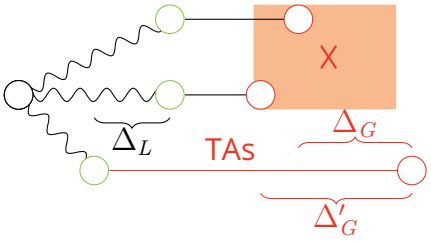
[1] We will provide a detailed overview of the existing definitions in Sec. 4.2 and of their underlying interpretations in Sec. 4.5.



(a) A concrete model (solid circles are states and solid edges are transitions) and a possible corresponding abstract model for static WCET analysis (dashed rectangles are over-approximated states, dashed edges and rods represent estimations).



(b) Abstract, static-analysis-centric interpretation: the TA is an artifact of abstraction combined with pruning ( ).



(c) Concrete, hardware-centric interpretation: the TAs impose hardware assumptions on analyses ( ).

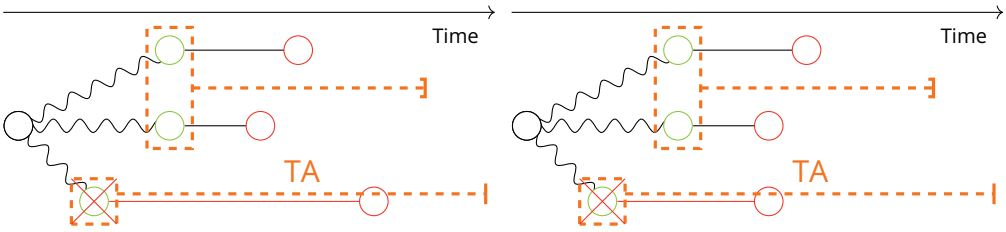
Figure 4.1: Two interpretations of TAs in the literature wrt. worst-case-execution-time analysis.

stract model (dashed shapes). The horizontal axis represents the time—needed to reach the states in the concrete model, and indicating the computed estimates in the abstract model. In the concrete model, the initial state (the leftmost circle) has three successor states (the solid green circles) that represent comparable microarchitectural states defining a latency—e.g., the corrugated transitions ( ) represent an initial instruction fetch under various cache contents—which in turn, for simplification, have one successor each as a final state. The final states (the solid red circles) also represent comparable microarchitectural states—e.g., the end of the global execution. In the corresponding abstract model, the two uppermost successors of the initial concrete state are merged into a single state (i.e., the uppermost dashed UPSacOrange rectangle)—this is an over-approximation—from which one (provisional) estimate of the global time can be computed ( ).

Since certain states or estimates are comparable, we can define *variations*. Fig. 4.1b shows the local variation Δ_L and the global variation Δ_G that can be identified in the abstract model. These variations form the typical situation of a counter-intuitive TA. Similarly, Fig. 4.1c shows the identical local variation Δ_L, as well as the two global variations Δ_G and Δ'_G that can be identified in the concrete model. Both pairs of local/global variations also represent counter-intuitive TAs in this model.

Note that variations may concretely occur only from distinct initial hard-

ware states, since the system, i.e., a processor and memories, is deterministic (cf. Sec. 1.1.3)—a given program executed from the same initial hardware state will always result in the same hardware behavior. The models in this section adopt a simplified viewpoint, with only one initial state of a *restricted* part of the microarchitecture (i.e., related to the input program and data) but a non-deterministic transition (⇝) to represent distinct behaviors, according to initial conditions that are not explicitly part of the models (e.g., depending on the contents of the caches).[2] In this way, we intend to clearly indicate that we consider various paths that are suitable for assessing TAs. We believe that the intuitive notion of TAs demands that the restricted initial states in our models should correspond to the components that are visible in the ISA (cf. Sec. 1.1.1). Thus, all the elements to which the code sequence refers must be part of the initial states, as well as the instruction memory where the code sequence is loaded. On the contrary, any element present in the implementation only (e.g., a cache) constitutes other initial conditions that may participate in variations. One can formally define the part of the microarchitecture that is retained for initial states and distinguished from other initial conditions, e.g., a "timing-relevant dynamic computer state for a program scope" [23].



(a) With a concrete TA. **(b)** Without concrete TAs.

Figure 4.2: A concrete model (solid shapes) and a possible corresponding abstract model (dashed shapes) that assumes a TA though the particular WCET estimate (⊣) obtained after pruning (⋈) is safe.

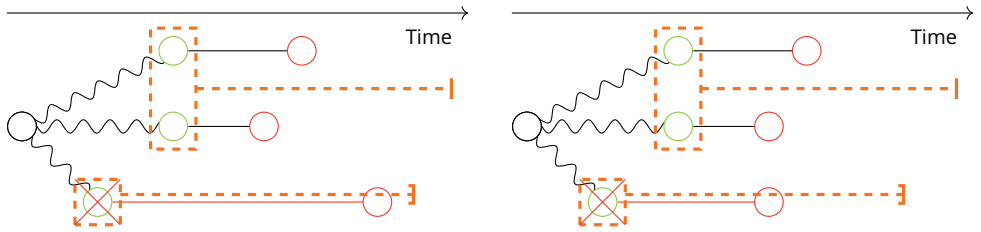
4.1.2 . Static-Analysis-Centric Interpretation

A first interpretation, based on abstract models, considers TAs to be an artifact of the analysis in combination with pruning [21, 22]. On complex hardware, despite the over-approximations, exploring the abstract model inevitably leads to state explosion, and thus, it would be desirable if some abstract states could be pruned during the analysis. According to the intuition, the pruned states might be those that represent favorable cases [ibid.] (e.g., cache hits), thus local speedups (Δ_L) wrt. another execution trace (cf. Fig. 4.1b). The estimate of the global execution

[2] Our simplified "concrete" models thus contain a certain degree of abstraction, but stand for reference models.

time *of a trace* (-|/-|) is always safe wrt. to the actual execution time of the (concrete) trace (see the solid red circles), i.e., for this trace in isolation—we assume that static analyses compute a sound upper bound from one abstract state, possibly with penalties that introduce pessimism only (Fig. 4.1a). However, this estimate is not necessarily safe wrt. *other* traces. An estimate obtained with pruning (from the abstract state that is preserved) may thus be unsafe. As pruning should also preserve the soundness of the WCET analysis, in this interpretation, a TA is identified for a given abstraction when a WCET estimate obtained with pruning is smaller (Δ_G) than without pruning, and it is safe to prune abstract states if they do not trigger TAs.

Note that under this interpretation, TAs may be assumed even when the WCET estimate obtained with pruning is safe wrt. the concrete hardware. For instance, assume that in a given abstraction, we get the abstract model shown in Fig. 4.2a for the unchanged concrete model, such that the particular time estimate (-|) obtained with pruning is larger than in Fig. 4.1b. The TA still exists in the abstract model. However, even with pruning and in spite of the actual TA in the concrete model, the so derived WCET estimate is safe, since it is an upper bound of the global execution time of any concrete trace—in particular, the trace pruned by the analysis (i.e., reaching the lowermost solid red circle). TAs may even be assumed when the concrete model does not contain TAs (i.e., a false positive in the detection of TAs). For instance, the abstract model considered in Fig. 4.2b may also correspond to the different concrete model represented in Fig. 4.2a. In this case, the concrete model has no TA and the abstraction introduces a TA. In both cases of Fig. 4.2, pruning is supposed to be harmful due to the TA, though the consequent WCET estimate would be safe.



(a) With a concrete TA. **(b)** Without concrete TAs.

Figure 4.3: A concrete model and a possible corresponding abstract model that indicates no TA though a particular WCET estimate (-|) from the pruned trace (⤫) would be still safe (and tighter).

Fig. 4.3 represents the two remaining combinations of presence/absence of counter-intuitive TAs in the concrete/abstract models. In Fig. 4.3, the abstract model indicates no TA. By definition, the WCET estimate obtained with pruning is *not* smaller than without pruning. However, as shown in Fig. 4.3, a particular WCET estimate (-|) obtained *without* pruning may be safe wrt. the concrete

hardware—and thus tighter than with pruning, whether there is a concrete TA (i.e., a false negative, Fig. 4.3a) or not (Fig. 4.3b).

The problem with this pure static-analysis-centric interpretation is that nothing makes it possible to discriminate between the tightness of the estimation method (e.g., with pessimistic penalties) used in a specific analysis and the issue that pruning could entail with the considered executions, even for arbitrarily accurate static analyses.

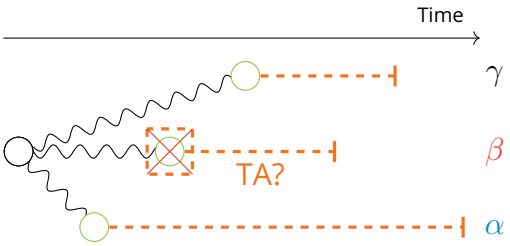
4.1.3 . Hardware-Centric Interpretation

TAs were originally *observed* in a real processor [20], outside of any analysis framework. This leads to another interpretation, based on the concrete model (Fig. 4.1c), where TAs originate from the hardware itself [80, 23, 24, 81]. This interpretation does not prevent from relating TAs with static WCET analysis. Pruning a trace during an analysis becomes unsafe (only) when the estimate without pruning is unsafe wrt. a concrete execution—namely, in Fig. 4.1c, where the two uppermost traces remain after pruning, when the analysis computes an estimate situated in the filled zone (▬). The absence of TAs in this interpretation thus indicates that *the hardware fulfills the underlying assumptions of a static analysis*.

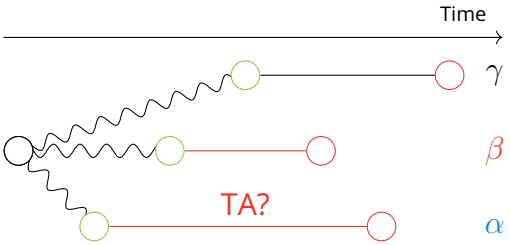
This second interpretation is also applicable outside of the traditional context of static WCET analysis. Notably, TAs may be problematic when trying to bound the impact of perturbations that may occur during the execution of a program. Preemptions or interrupts would be a typical example of such situations. The perturbations may lead to new hardware states that would not occur during an execution of the program in isolation, e.g., an instruction cache miss might occur even for two successive instructions on the same cache line. Let us assume that an analysis provides a safe estimate for the program in isolation (without perturbations). In the presence of a (concrete) TA, this estimate might become *unsafe* due to the impact of a perturbation on a pruned trace, even if pruning was safe during the static analysis, and even if the analysis introduces pessimistic penalties. Besides, TAs jeopardize other timing-analysis methods partially based on concrete executions, such as measurement-based analysis and probabilistic analysis (cf. Sec. 1.1.4). **Considering that TAs are primarily concrete execution phenomena and that their understanding from this perspective allows for better controlling the accuracy of static analyses and for integrating perturbations, we believe that the impact of TAs should be studied and understood *independently of the* WCET*-analysis technique itself*; we thus adhere to the hardware-centric interpretation of TAs**.

4.1.4 . Absolute-WCET Interpretation vs. Pairwise Interpretation

If we face at least three traces, a variation in a trace may trigger a counter-intuitive TA wrt. one or some of the other traces only, without impacting the identification of the (actual or estimated) WCET, so that the TA is harmless with regard to the WCET. Certain authors consider that such harmless TAs, where the worst case



(a) Analysis-centric interpretation: β is pruned due to γ, thus the TA between α and β is not captured.



(b) Hardware-centric interpretation: the actual WCET is given by γ, thus the TA between α and β is not captured.

Figure 4.4: Three execution traces α, β and γ, in which α and β form a TA in the *pairwise interpretation*, while the existence of γ hides the TA in the *absolute-*WCET *interpretation*.

is not directly involved, do not even constitute a TA [79, 22, 81]. This restrictive *absolute-*WCET interpretation, where only certain traces can be involved in a TA, is in contrast to the *pairwise* interpretation, where any pair of execution traces may trigger a TA [80, 23, 21]. Nevertheless, the identification of such traces differs according to whether the analysis-centric or the hardware-centric interpretation is adopted. In the former case, the actual WCET must be impacted, and the final WCET estimate from a systematic pruning method in the latter.

The absolute-WCET interpretation, which naturally often relies on a static-analysis method, and the analysis-centric interpretation, which intrinsically rests on pruning, make more sense together.[3] A TA then occurs only if a remaining (abstract) trace *after maximal pruning* is involved, thus compromising the WCET bound. Fig. 4.4a illustrates a situation where the combined absolute-WCET and analysis-centric interpretations hide a TA. Note that β does not trigger a TA wrt. γ and[4] that, on the contrary, α triggers a TA wrt. γ, even in the restrictive absolute-WCET interpretation. However, trace α triggers a TA wrt. β in the pairwise interpretation yet not in the absolute-WCET interpretation, since β is removed

[3] In practice, the analysis-centric interpretation is often often associated with the absolute-WCET one, and conversely (cf. Table 4.2).

[4] Even in the pairwise interpretation; otherwise, it would also in the absolute-WCET interpretation (isolated estimates are sound).

after pruning (only γ remains). This reinforces the fact that the static-analysis-centric interpretation (in combination with the absolute-WCET interpretation) allows verifying the assumptions of a certain static-analysis method (with pruning) rather than strictly reasoning about TAs.

The hardware-centric interpretation is not related to a particular static-analysis method. Thus, in this case, the absolute-WCET interpretation restricts TAs to situations where a trace leading to the *actual* WCET is involved. Fig. 4.4b represents a situation where the absolute-WCET and hardware-centric interpretations hide a TA. Trace α triggers a TA wrt. β in the pairwise interpretation yet not in the absolute-WCET interpretation, since it does not trigger a TA wrt. γ, which leads to the actual WCET. Note that the exact situation in Fig. 4.4b can also be illustrated with an abstract model (thus in the analysis-centric interpretation); in this case, the trace that remains after pruning (i.e., γ) is also the trace yielding the WCET bound.

We intend to study TAs independently of any WCET analysis technique (as mentioned above), and, moreover, the actual worst case is likely to be inaccessible (cf. Sec. 1.1.3). The WCET is in any case not known a priori—and may be impacted by perturbations—, whereas the restrictive absolute-WCET interpretation might be useful only if some traces unequivocally did *not* constitute the worst case. **We thus believe that TAs should be defined from *pairs* of (concrete) execution traces**, e.g., trace α in Fig. 4.4b triggers a TA *wrt.* trace β (whereas it does not wrt. γ).

4.2 . Overview of the Definitions of TAs

In this section, we provide an overview of the existing definitions of TAs. Lundqvist and Stenström first introduced the notion of (counter-intuitive and amplification) TAs [20], from the observation of different behaviors of a processor when executing the same program, namely (execution) traces. Their semi-formal definition is based on instruction sequences whose first instruction has a variable latency, e.g., due to a cache hit/miss. They define the notion of TAs by comparing two execution traces and provide examples for an OoO processor. *The definition is incomplete as it only allows for a* single *latency variation at the first instruction and does not define instruction latencies.* Wenzel et al. adopt the same framework [21]. Though latencies are clearly defined as the time spent in functional units, their definition is still restricted to a single instruction variation. The introduced "resource allocation criterion" that provides a necessary condition for the occurrence of TAs thus *cannot accommodate with more variations or with* TA*s that do not primarily originate from the scheduling* (cf. Sec. 1.3.1). Moreover, the definition demands "almost identical" initial hardware states, without a clear definition.

Mainly semi-formal definitions [20, 21] exist for amplification TAs, and to our knowledge, only Kirner et al. [80, 23] provide formal definitions of such TAs

(also called weak or strong-impact anomalies). In the following, we focus on *counter-intuitive* TAs, which are dominant in the literature, and we consider *formal* definitions, for which we were able to develop formal and executable models (cf. Ch. 5).

The formal definitions of counter-intuitive TAs mainly differ in two orthogonal features: the various *interpretations* of the TA phenomenon against the WCET, and the essential notion of *variations*. We already mentioned that the source of TAs is subject to various interpretations (cf. Sec. 4.1). The definitions often assume a "hardware model" or an "abstract hardware model", without more details, whose refinement level might thus be subjective. Consequently, the very definitions of TAs often do not directly involve the nature of the hardware model (concrete or abstract), and the underlying positioning wrt. static WCET analysis remains an ancillary interpretation of the definitions. However, the restrictive absolute-WCET interpretation (cf. Sec. 4.1) may be apparent.

The definitions differ in the way of defining latencies and then (local and global) variations that entail TAs. We found four typical ways of comprehending variations in the literature, around which we structure the following overview. However, **the definitions are often incomplete and only illustrated through partial examples** (e.g., scheduling diagrams as in Fig. 2.4b). When necessary, we highlight problems that we encountered while trying to encode the definitions in a systematic manner. In Ch. 5, we will establish assumptions that enabled us to overcome these problems and to encode the definitions.

4.2.1 . Step Heights in Step Functions

A simple definition of TAs is provided by Gebhard [24]. In this definition, execution time $\gamma(\eta, i)$ assigns the i-th instruction of a sequence its latency, depending on the initial hardware state η. Initial hardware states may cover, for instance, the cache content. The global execution time up to instruction n, from initial state η, is the sum of the execution times of each instruction: $\Gamma(\eta, n) = \sum_{i=1}^{n} \gamma(\eta, i)$. Then, counter-intuitive TAs are defined from these notions. The formal definition mixes words with mathematical symbols. However, according to our understanding, the definition can be stated as follows; a TA occurs for a given input sequence of instructions iff:

$$\exists \theta,\ i < n,\ \forall \eta \neq \theta,\ \gamma(\theta, i) < \gamma(\eta, i)\ \wedge\ \Gamma(\theta, n) \geq \Gamma(\eta, n) \tag{4.1}$$

where θ and η are two initial hardware states. TAs are thus defined by the existence of an initial state θ, called anomalous, that leads to the shortest execution time (i.e., a local speedup) for the i-th instruction and that eventually results into the largest global execution time at the n-th instruction (i.e., a global slowdown).

Firstly, note that the non-strict inequality includes borderline cases in situations that constitute a TA, where another initial state leads to the *same* global execution time as the anomalous initial state. We consider that this inequality should be turned into a strict inequality to fit the intuitive understanding of TAs introduced

in Sec. 1.3.1. Secondly, though intuitive, this definition restricts counter-intuitive TAs to the situations where, in a trace starting from an anomalous initial state:

1. the global execution time at the point of a later instruction (n) is *the largest* one, i.e., the WCET;
2. and the latency of the incriminated instruction (i) is *the shortest* one

against *all* initial hardware states. Whereas the first item rigorously corresponds to the absolute-WCET interpretation discussed in Sec. 4.1 that excludes harmless cases wrt. the WCET, we cannot find reasons for the second item, since in *any* interpretation, at least one trace that is responsible for a TA may originate from any initial hardware state, and, moreover, the trace with the shortest latency is never taken as a reference. We thus believe that the above formula should be adapted at a minimum to exclude borderline cases wrt. the WCET and to have both (local and global) comparisons independent:

$$\exists \theta, \eta,\ i < n,\ \forall \tilde{\eta},\ (\gamma(\theta, i) < \gamma(\eta, i))\ \wedge\ (\Gamma(\theta, n) > \Gamma(\tilde{\eta}, n)) \tag{4.2}$$

Moreover, since we believe that TAs should be defined from *pairs* of traces (cf. Sec. 4.1), we could also modify the first item, related to the absolute-WCET interpretation, and simplify the adapted formula, in order to provide a sound formal definition, in the pairwise interpretation, that nonetheless remains consistent with the spirit of the paper [24] as to latencies and variations. A TA would occur iff:

$$\exists \theta, \eta,\ i < n,\ \gamma(\theta, i) < \gamma(\eta, i)\ \wedge\ \Gamma(\theta, n) > \Gamma(\eta, n) \tag{4.3}$$

The derived definition (based on the last formula) **still leaves several details open**. For instance, it relies on the notion of hardware states without a clear definition. The same applies to instruction latencies, which are only supposed to be non-negative and yield the execution time when summed. Instruction latencies are obvious on in-order processors, but the situation is more complex for OoO processors. The notion of latencies used by Wenzel et al. [21] (i.e., the time spent in FUs), for instance, is not admissible due to the second constraint.

Note that Reineke and Sen proposed another definition [17] that is presented as a relaxed version of the one based on instruction locality (cf. Sec. 4.2.4). This definition relies on the same kind of step functions, which are nevertheless explicitly obtained from the instants when instructions start execution (i.e., when they are fetched), so that the step heights represent the duration between two states that correspond to the fetch operations of two successive instructions. However, this definition is **limited to timing variations at the first instruction of the sequence**. Besides, the definition adopts a simplified absolute-WCET interpretation, in which only the longest traces are considered among several traces that share the same states up to (and including) the variation, but that diverge after the state where the second instruction is fetched.

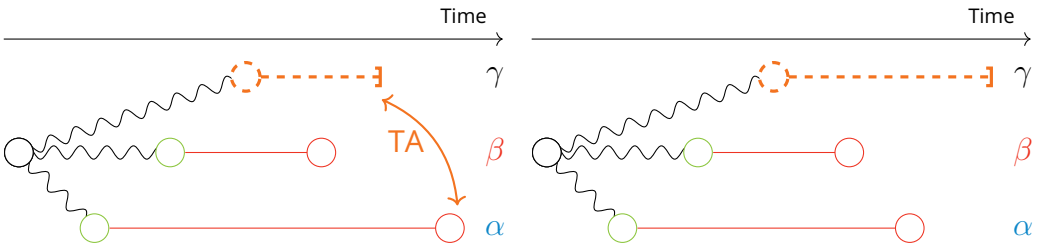
4.2.2 . Intersections in Step Functions

An alternative definition of TAs, which also relies on the cumulative execution times, was proposed by Kirner et al. [80]. This definition is *exclusively* based on cumulative execution times, i.e., it focuses on the instants when the execution of each instruction completes. Keeping the same notations as in the previous section, a TA occurs iff:

$$\exists \theta, \eta,\ i < n,\ \Gamma(\theta, i) < \Gamma(\eta, i)\ \wedge\ \Gamma(\theta, n) > \Gamma(\eta, n) \tag{4.4}$$

Cassez et al. [81] proposed a similar definition, with the notion of "consistently as slow" hardware states, yet in the restrictive absolute-WCET (though also hardware-centric) interpretation that excludes harmless TAs wrt. the WCET (see Fig. 4.4b). Still with the same notations, a TA thus occurs iff:

$$\exists \theta, \eta,\ i < n,\ \forall \tilde{\eta},\ (\Gamma(\theta, i) < \Gamma(\eta, i))\ \wedge\ (\Gamma(\theta, n) > \Gamma(\tilde{\eta}, n)) \tag{4.5}$$



(a) A TA is stated. **(b)** No TA is stated.

Figure 4.5: Eisinger et al. [79] mix the hardware-centric interpretation with the static-analysis-centric interpretation, leading to a particular absolute-WCET interpretation (cf. Sec. 4.1).

Another definition of TAs is proposed by Eisinger et al. [79]. In the pairwise interpretation, this definition would be equivalent to the other two variants, with the only difference that the two axes of the step function are switched, i.e., it tracks the number of instructions completed in an arbitrary time window. However, the authors actually interpret TAs in a particular manner that combines the hardware-centric interpretation with the static-analysis-centric interpretation. They indeed base their definition on a specific reference trace obtained from a *fictive* behavior that is supposed to represent the abstract model used in a WCET analysis *after pruning*. A TA is then defined in a particular absolute-WCET interpretation, as a situation where the execution time of any *concrete* trace is larger than that of this abstract reference trace. In sum, this situation might correspond to the filled zone in Fig. 4.1c, where, nevertheless, it would *define* a TA,[5] whereas this situation

[5]And where the reference trace would always be obtained with maximal pruning.

invalidates a hardware assumption in a pure static-analysis-centric interpretation (cf. Sec. 4.1). Fig. 4.5 illustrates this interpretation in further detail. Fig. 4.5a echoes Fig. 4.4a; here, pruning serves for constructing the (single) abstract trace only. A TA is stated between α and γ (and not between α and β), exactly as in Fig. 4.4a. Similarly, Fig. 4.5b echoes Fig. 4.4b. No TA is stated, and the TA between α and β is not captured, exactly as in Fig. 4.4b. The approach for defining TAs remains WCET-centric and does not enable to strictly reason about TAs. However, this particular absolute-WCET interpretation has the advantage of eradicating the artifacts caused by abstraction, over pure static-analysis-centric interpretations that fully rely on abstract models (see Fig. 4.2).

4.2.3 . Component Occupation

The previous definitions summarize the timing of individual instructions using a single value, the instruction latency. Kirner et al. [23] propose a different view that focuses on the use of a hardware component (resulting from a partitioning of the whole microarchitecture) throughout a trace. By comparing the amount of time when a FU is occupied, among two traces, along with the execution times of the traces, they describe a new type of TAs called *parallel inversion*.

The use of FU_1 for the traces depicted in Fig. 2.4b and 2.4c, for instance, amounts to 4 and 6 cycles for α and β, respectively. Despite the higher use of FU_1, the execution time of β is lower, which indicates a TA. The use of FU_2 is the same in both traces and thus does *not* indicate a TA. A component including both FUs again yields a TA.

The main problem with this definition is that **the authors [23] do not describe what a suitable hardware component is**, e.g., whether/how FUs have to be grouped together in a single component. As illustrated by the example from above, the hardware partitioning has an impact on the identification of TAs. Unfortunately, the authors do not describe how to obtain partitions that reliably identify TAs.

4.2.4 . Instruction Locality

Reineke et al. [22] propose another point of view, making it possible to combine per-instruction latencies with the notion of occupation of resources—the so-called *locality*. It is based on a transition system that specifies the *cycle-level* behavior of the considered processor. The authors assume that one can derive an assignment of instructions to resources (hardware components) from a given state of the transition system. Consequently, it is possible to extract, on every cycle, the locations (e.g., a FU) occupied by an instruction; this is called a *locality constraint*. The authors also assume that the occupation of locations by an instruction may change due to non-deterministic behavior that represents unknown information at the analysis stage. TAs are then defined by comparing the occupation of locations *by an instruction* at the first instant when execution traces diverge, along with the global execution times of the traces.

This definition is basically related to static WCET analysis. It is based on the analysis-centric, absolute-WCET interpretation, illustrated by Fig. 4.4a. The authors consider the decomposition of trace π under locality constraint l as a prefix, local sub-trace $\pi_{|l}$ in which l holds, and a suffix: $\pi = \pi_{\text{pre}} \circ \pi_{|l} \circ \pi_{post}$. Thus, whenever some traces share the same prefix and then diverge due to a local variation at a given locality constraint, this definition identifies the variation with the largest latency as a local worst case. A *local-worst-case* trace corresponds to a trace that always follows the local worst cases (i.e., for all variations): trace π is a local-worst-case trace iff, for any trace π' and any locality constraint l, whenever π and π' share a common prefix π_{pre} so that $\pi = \pi_{\text{pre}} \circ \pi_{|l} \circ \pi_{post}$ and $\pi' = \pi_{\text{pre}} \circ \pi'_{|l} \circ \pi'_{post}$, then $\left|\pi_{|l}\right| \geq \left|\pi_{|l'}\right|$ (otherwise it is a non-local-worst-case trace). In Fig. 4.4a, the prefix is restricted to the initial state and the locality constraint that triggers the (single) variation to one state (in green) for each trace; thus, γ is the local-worst-case trace. Then, this definition identifies a TA when the WCET bound (given by α in Fig. 4.4a) is not derived from a local-worst-case trace (but from a non-local-worst-case trace); hence the TA in the figure.

A first problem with this definition concerns the interpretation of TAs. The underlying abstract transition system is abstract and cannot be a faithful model of the actual hardware (cf. Fig. 4.2). It is unclear how to obtain suitable abstractions in practice. **A second, more tangible problem concerns the locality constraints, which are described as *convex predicates*.** The authors do not explain how to chose suitable locations (only that they should be at the pipeline-stage level), nor how to obtain corresponding convex predicates. Thus, it is unclear how to compare the occupation described by a locality constraint once the traces have diverged, e.g., when an instruction occupies different locations in the two traces in the next cycle. It is also unclear whether it is always possible to identify a local-worst-case trace—the condition introduced above should hold for *any* locality constraint. In the event that it is not possible, the definition indicates the *existence* of a TA, since the WCET bound is indeed *not* derived from a local-worst-case trace. Conversely, from the above criterion, traces are local-worst-case by default, if there is not a common prefix. By construction, the definition indicates the *absence* of TAs if no non-local-worst-case trace is identified and, consequently, the WCET bound is given by a local-worst-case trace. This introduces a surprising asymmetry between the two borderline cases. Note that locality constraints would raise issues for comparing traces even in the pairwise interpretation.

4.3 . Predictable Cores

Lundqvist and Stenström [20] proposed the utilization of synchronization instructions as a software counter-measure for removing TAs emerging from an OoO scheduling. However, they did not precisely explain where these instructions should be placed to guarantee the absence of such TAs.

Specific cores based on hardware counter-measures have been designed so as to inherently improve the timing predictability of the hardware, by enforcing a regular timing behavior and, desirably, removing or limiting TAs. **All these cores prevent execution scenarios that are known to entail TAs, but no *definition* of TAs per se is provided.** Hereafter, we introduce a few predictable cores (Sec. 4.3.1), as well as a specialized abstraction for analyzing such hardware designs wrt. amplification TAs (Sec. 4.3.2).

4.3.1 . Specific Hardware Designs

Rochange and Sainrat [82] implement specific hardware mechanisms to regulate the fetch of basic blocks on OoO pipelines supporting an instruction-prescheduling policy. They actually fetch the instructions of a basic block only when they cannot induce stalls within the basic block, thus concentrating the impacts of variations within the basic blocks. Their goal is not to strictly remove TAs, but instead to provide a flexible mechanism to achieve a trade-off between performance and predictability (cf. Ch. 1). Whitham and Audsley [83] continue this work, improving instruction-level parallelism by resynchronizing sequences of basic blocks instead of basic blocks, which tend to contain few instructions.

The absence of counter-intuitive TAs is a prerequisite for a regular timing behavior. The following cores are simple—they widely exclude OoO execution, well known to cause counter-intuitive TAs—but they implement counter-measures to specifically limit amplification TAs in addition.

The PRET microarchitectures [84] are designed to perform *repeatable* executions, thus achieving predictability at source, by removing most features that may introduce non-determinism—and time dispersion—when executing a program (cf. Sec. 1.1.3). An example of such a microarchitecture is PTARM [85], ensuring repeatability through a thread-interleaved pipeline, a scratchpad memory, and a specific DRAM controller. FlexPRET [86] adapts this principle for mixed-criticality applications, providing guarantees for hard real-time threads but allowing soft real-time threads (cf. Sec. 1.1.2) to execute against the predictability, in order to globally improve performance. These cores thus purely disable the features that we consider (e.g., caches), to avoid TAs.

Although amplification TAs are often ignored or seen as a mere variant of counter-intuitive TAs (cf. Sec. 4.2), Hahn et al. [19] analyzed amplification TAs and their link with compositionality. They highlighted the typical situation introduced in Sec. 2.1.2, where an amplification TA may occur due to memory interference, (even) in in-order pipelines. However, they did not provide a formal definition of amplification TAs, nor did they rely on any established formal definition. The Patmos core [87] decouples the memory accesses for instructions and data, thus preventing the memory-interference scenario illustrated in Sec. 2.1.2 [26]. Hahn and Reineke continue the work on compositionality [19] with the development of a pipeline called SIC [25], designed to be free from TAs. The SIC core targets in particular TAs that occur due to the mentioned memory-interference scenario,

this time by delaying the incriminated instruction-fetch memory accesses.

All these in-order cores are based on simple pipelines and it is unclear how to extend the results to common OoO pipelines, e.g., based on Tomasulo's algorithm [38]. MINOTAuR [88] extends the framework of SIC to a more complex microarchitecture that allows speculative execution and can execute *independent* instructions out of order, nevertheless stalling decoding on dependencies and then respecting program order on functional units.[6] Similarly, Vicuna [89] is a timing-predictable vector co-processor based on SIC and relying on the same hardware counter-measure.

4.3.2 . Canonical Model for Assessing Compositionality

Jan et al. [26] use model checking in order to prove the absence of amplification TAs (which hinder compositional analyses) in predictable pipelines and compare their hardware approach to avoid such TAs. This work assumes a prerequisite for TAs, in the particular situation of memory interference (cf. Sec. 2.1.2), and does not aim at providing a definition. The authors propose a particular abstraction, called *canonical pipeline model*, and use it to verify the absence of such anomalies on PRET [84], Patmos [87], and SIC [25]. In addition, they identify TA scenarios for the K1 pipeline [90]. **The canonical model serves as a basis for our formal modeling of TriCore** (in Ch. 9).

Hardware Abstraction

The canonical model is based on a hardware abstraction addressing the *timing behavior* of the considered processors only; consequently:

- Instruction variants are not distinguished individually, only instruction classes such as load/store or arithmetic/logic operations.
- Computations (i.e., arithmetic) are not modeled, only the pipeline logic that may impact how instructions progress through pipeline stages as well as the interactions with caches and the external bus.
- Cache content is not modeled, only the (potential) impact of hits and misses.
- Only the interactions between *two* instructions, i.e., the $downstream$ instruction and the $upstream$ instruction, are modeled explicitly, side-effects of other instructions in the pipeline are over-approximated.

The *downstream* instruction precedes (not necessarily directly) the *upstream* instruction in the flow of instructions. Therefore, the downstream instruction (older) is more advanced in the pipeline, i.e., in later stages of the pipeline, compared to the upstream instruction (younger).

From a structural point of view, the downstream instruction may advance through the pipeline, provided that the memory bus is not busy if the instruction

[6] This design implements *scoreboarding*, a restricted form of OoO execution.

has to access main memory. Hence, although the downstream instruction precedes in the instruction flow, it may be delayed because of a shared resource (i.e., the bus prompts a structural hazard). This is actually a source of amplification TAs (cf. Table 2.2). The upstream instruction, besides being potentially stalled by a bus access, may also be prevented from advancing to the next stage, potentially occupied by the downstream instruction. Beyond these structural aspects, microarchitectures may implement stalling strategies that fully or partially stall the pipeline to enforce a more regular behavior. Such additional progression strategies have been modeled within their canonical pipeline model. In particular, we adapt two of their strategies in Part. IV: the *whole* logic stalls the entire pipeline as soon as any instruction induces a stall, whereas the *only-upstream* logic allows the downstream instruction to continue advancing through the pipeline while the upstream instruction is stalled.

The canonical model was encoded in UCLID5 (cf. Sec. 3.3.1). Each attribute of both instructions is represented by UCLID5 state variables and each transition (`next` block) corresponds to a clock cycle of the processor.

Verification Procedure

The canonical model is used to verify the pipelines using the BMC engine of UCLID5 (cf. Sec. 3.3.1). The basis of the verification procedure consists in initializing the bounded model checker such that the downstream instruction is placed in *any* stage of the pipeline ($stage$ attribute) and the upstream instruction is placed in the pre stage, i.e., about to be issued. This is actually equivalent to successively choosing all possible initial stages for the downstream instruction when the upstream instruction enters the pipeline, as represented in Table 4.1. This table exemplifies the initial placement and the progression of the upstream instruction, as well as of *successively* explored downstream instructions corresponding to *different initializations* of the current stage (in the same 5-stage in-order pipeline as in Table 2.2). In this way, all possible distances between instructions in the pipelines are evaluated. We note that not all possible execution behaviors, i.e., baseline latencies, are represented in the table. All classes of instructions and all possible baseline latencies are exhaustively explored by model checking. The state space is completely *determined* after the initialization step (`init` block), through assumptions (see Sec. 3.3.1) that permit the model checker to arbitrarily choose the initial values of variables representing the initial pipeline stage of the downstream instruction, the classes of both instructions, and the values of baseline latencies of both instructions for the stages that may access memory—thus capturing local timing variations.

4.4 . Timing Modeling in Pipelines

In this section, we introduce the related work on timing modeling for the purpose of timing analysis. We introduce modeling approaches aiming towards model checking

Table 4.1: Example of successive downstream instruction placements explored by the model checker through the arbitrary choice of the initial stage, in a 5-stage in-order pipeline. The upstream instruction is the one about to enter the pipeline (pre stage) at the initial clock cycle (t_0).

Cycle	t_0	t_1	t_2	t_3	t_4	t_5
...	...					
Down. 3	EX	MEM	WB			
Down. 2	ID	EX	MEM	WB		
Down. 1	IF	ID	EX	MEM	WB	
Upstream	pre	IF	ID	EX	MEM	WB

(Sec. 4.4.1) and towards specialized analytical methods (Sec. 4.4.2). We also introduce the related work on the timing modeling of TriCore (Sec. 4.4.3).

4.4.1 . Model Checking of Timing Properties

In Ch. 3, we mentioned model-checking applications that aim at verifying functional properties, either on hardware or software. Hereafter, we cite model-checking applications that aim at verifying *timing* properties, taking into account the combination of hardware and software in real-time systems.

WCET Analysis

Metzner [91] showed that model checking can be applied in the context of WCET analysis (cf. Sec. 1.1.4), nevertheless with a limited hardware model comprised of an ideal pipeline—thus without TAs—whose timing behavior is modeled through pre-estimated costs. Huber and Schoeberl [92] adopted a similar approach, advocating for combining model checking with traditional WCET methods such as IPET (cf. Sec. 1.1.4). Dalsgaard et al. [93] and Gustavsson et al. [94] used model checking to determine the WCET on simplified but more realistic in-order-hardware models. Metta et al. [95] proposed abstractions at software level to improve the scalability of WCET analyses by model checking. None of these methods primarily takes TAs into consideration; the model of Dalsgaard et al., for instance, could accommodate the occurrence of TAs when computing WCETs, but this is not the focus of this work and the authors report that their methods suffer from state explosion [93]. Furthermore, we do not basically seek to estimate or check WCETs, but to analyze TAs, possibly in a broader context than that of WCET analyses.

Detection of TAs

It follows from Sec. 4.2 that the definitions show issues that make it *difficult to put them into practice* in order to detect TAs on realistic models. Consequently, the definitions are most of the time not implemented as procedures. The definition by Eisinger et al. [79] (cf. Sec. 4.2.2) is an exception, since it is accompanied with a model-checking procedure to detect TAs automatically on the model of an OoO processor; however, they do not provide details about their formal model and they adopt a absolute-WCET interpretation. Asavoae et al. [34, 96] also use

model checking in a first attempt to make the definition by Reineke et al. [22] (cf. Sec. 4.2.4) executable; this work assumes a relaxed version of this definition targeting the execution stage of an in-order or OoO pipeline, and shows how it can be auspiciously integrated into an automatic tool.

4.4.2 . Analytical Methods

With the design of SIC (cf. Sec. 4.3.1), Hahn and Reineke [25] base their analysis on a monotonicity property resting on a progress notion [18], as a sufficient condition for the absence of TAs. This work does not make a clear distinction between both types of TAs and uses monotonicity, intuitively related to *counter-intuitive* TAs, to ensure compositionality (related to amplification TAs) in the case of the *specific* design of SIC. Moreover, though the essential notions of progress and then monotonicity are natural, they are specialized for classical in-order pipelines, and it seems difficult to transpose them to an OoO context.

In Ch. 7, we propose a novel definition of counter-intuitive TAs, based on a notion of causality between events occurring in a pipelined execution. Our event-style approach for modeling timing dependencies in a pipelined processor is similar to that of Li et al. [35] (more details are given in Ch. 7). They use *Execution Graphs* (EG) that model the timing semantics of an OoO pipeline through events, latencies, and the imposed order between events, in order to provide analytical WCET estimates. Bai et al. [97] extend the EG with a more compact but equivalent symbolic data structure called *Execution Decision Diagram*. Hahn et al. [19] use *microarchitectural execution graphs* to represent possible durations between events of interest. However, they focus on abstract states that do not permit distinguishing the effects of individual instructions nor the resource use needed to identify causal relationships.

4.4.3 . Timing Modeling of TriCore

In Ch. 9, we propose a formal model of the TriCore microarchitecture introduced in Sec. 2.2 to study amplification TAs. Nguyen et al. [98] also propose a formalization of TriCore that is then explored, using model checking, towards identification of memory interference. Our model differs from that of this work in the important aspect that we consider a finer timing granularity, as we propose a core-level microarchitecture modeling (as opposed to an inter-core model). Our formal investigation of amplification TAs is a prerequisite to the analysis in this work. Another compositional timing analysis is presented by Wilhelm and Wachter [99], sharing the same TriCore pipeline timing granularity as ours. However, to implement a symbolic pipeline analysis, modeling the fetch and decode stages is sufficient [99], while tracking amplification TAs requires a modeling of the temporal behavior of all pipeline stages.

A variant of the TriCore dual-pipeline is used by Ungerer et al. [100] in the time-predictable multicore platform named Merasa. To the best of our knowledge, the predictability aspects of a Merasa core are not formally investigated

using formal methods. Finally, the TriCore microarchitecture is considered by Sun et al. [101] wrt. its integration into a WCET analyzer called Otawa [102]. The analyzer provides a description language that allows custom microarchitecture designs to be plugged into the timing analyzer of Otawa. This work describes the TriCore microarchitecture (i.e., I, LS and loop pipelines introduced in Ch. 2) in the microarchitecture description language, providing accurate timing information. Our TriCore model considers similar timing parameters, but is also integrated into a formal specification and verification framework.

4.5 . Synthesis of the Various Definitions of Timing Anomalies

We introduced the various interpretations of TAs that can be found in the literature. We believe that TAs must be basically understood independently of WCET-analysis techniques, which are however sound only if they comply with the potential TAs. **Thus, we consider that TAs arise from the comparison of two execution traces on a concrete hardware model.**

Table 4.2 provides a synoptic view of the existing definitions of TAs. It reports, for each definition, the key features for defining *variations* (the guideline of Sec. 4.2), as well as their *interpretations* against the WCET. Since amplification TAs are never formally defined alone, but in a similar way to counter-intuitive TAs[7], we also report whether the related work provides a definition of amplification TAs in addition to counter-intuitive TAs.

Many definitions are based on hardware models restricted to theoretical concepts (as the generic form of a transition system) that do not refer to the actual microarchitectural components. Consequently, the papers remain theoretical. We report the exceptions in Table 4.2: the definition by Eisinger et al. [79] is accompanied with a detection procedure and Asavoae et al. [34, 96] proposed a detection procedure based on a simplified version of the definition by Reineke et al. [22] (cf. Sec. 4.4.1). Only Jan et al. [26] implemented a detection procedure for (a prerequisite of) amplification TAs (cf. Sec. 4.3.2).

[7]The provided definitions of amplification TAs are very similar to their counterparts for counter-intuitive TAs (translating the intuitive understanding introduced in Ch. 1), since the definitions of both types share the same frameworks and the same underlying definitions of variations. Note that this is facilitated by the fact that the counterparts in question are not WCET-centric.

Table 4.2: Synoptic view of the existing definitions of TAs, reporting the retained criteria for defining *variations* (cf. Sec. 4.2) and the *interpretations* (cf. Ch. 1) regarding the positioning in relation to WCET analysis—the symbol "?" stands for undetermined and "∼" for simplified version.

	Time in FUs	Step heights (Sec. 4.2.1)	Intersections (Sec. 4.2.2)	Global occupation (Sec. 4.2.3)	Locality (Sec. 4.2.4)	Analysis-centric	Absolute-WCET	Pairwise	Hardware-centric	Version for amplification TAs	Detection procedure (Sec. 4.4.1)
Definitions	Variations					Interpretations					
Lundqvist and Stenström [20]	?							✓	✓	✓	
Wenzel et al. [21]	✓					✓		✓		✓	
Gebhard [24]		✓					?		✓		
Reineke and Sen [17]		✓				✓	∼				
Kirner et al. [80]			✓					✓	✓	✓	
Cassez et al. [81]			✓				✓		✓		
Eisinger et al. [79]			✓			✓	✓		✓		✓
Kirner et al. [23] (*parallel*)				✓				✓	✓	✓	
Reineke et al. [22] – *[34, 96]					✓	✓	✓				∼*

PROBLEM STATEMENTS

OUR ultimate goal consists in disposing of reliable detection procedures of TAs, which are essential to ensure timing predictability in real-time systems. **A first problem concerns the identification of a reliable definition of counter-intuitive TAs.** Detection procedures must be based on formal definitions to decide on the presence or absence of TAs. Table 4.2 shows that the existing definitions of counter-intuitive TAs adopt combinations of *interpretations*; consequently, the results of hypothetical detection procedures based on various definitions are not comparable from this feature. However, we can bring the definitions down to a common interpretation and then build detection procedures. We thus face procedures based on various criteria for defining variations (Table 4.2).[8] This raises the question: *Do procedures based on various definitions entail the same results about the presence or the absence of counter-intuitive* TA*s?* **In Part II, we present the implementation of such detection procedures and we address this issue, which results in the proposal of a novel definition of counter-intuitive TAs in Part. III.**

A second, complementary issue concerns the formal modeling methodology and the verification strategy. We illustrate this problem from the detection of amplification TAs, though the methodology could be extended to counter-intuitive TAs. Jan et al. [26] applied their detection procedure only to simple predictable pipelines (besides the textbook pipeline introduced in Sec. 2.1, as a reference), in which the actual absence of amplification TAs allows for compositional analyses. In contrast to the previously studied architectures, we consider the complex, industrial TriCore microarchitecture (cf. Sec. 2.2), which actually suffers from amplification TAs. The consequence is two-fold and raises the following questions: *Can we provide suitable abstractions to comply with the sophisticated microarchitecture? How can we monitor and guide the verification engine towards covering the state space in desired ways, for the purpose of inserting countermeasures?* **In Part IV, we thus wonder whether and how amplification TAs can be tracked efficiently for such a microarchitecture.**

[8] Here, we focus on counter-intuitive TAs since all existing definitions address this class, sometimes with *similar* counterparts for amplification TAs (see Sec. 4.5).

Part II
Limitations of the Existing Definitions of Counter-Intuitive Timing Anomalies

In this part, we tackle the first problem introduced in Sec. 4.5, namely the impact of the various criteria for defining variations on the verdict about the presence/absence of counter-intuitive TAs (in the pairwise interpretation of TAs). We set up a **unified formal framework** for assessing the various definitions. This framework allows us to compare the verdicts of the definitions and to determine whether one definition dominates the others and could serve as a reference for the study of TAs.

Applying a formal definition of TAs means encoding a detection procedure from this definition, based on a formal hardware model. The goal of this part is thus to put existing definitions to the test, by comparing the verdicts of the subsequent detection procedures for a representative OoO pipeline. Are those definitions able to capture the intuitive understanding of TAs (cf. Sec. 1.3.1)? Do those definitions provide reliable and coherent answers when applied to different execution scenarios? In order to answer those questions, we have:

1. **developed a parametric *formal* model (Ch. 5) of the OoO-pipeline template introduced in Sec. 2.3** [1, 2];
2. **encoded the most relevant formal definitions of TAs into *executable* procedures (Ch. 5) coupled to the processor model** [2];
3. **then assessed the definitions (Ch. 6) through model checking by finding examples that lead to contradictions** (among those definitions) [2].

We make this assessment independently of any WCET-analysis technique. Note that no definition presents restrictions on its application conditions (the definitions only require, generally speaking, a hardware model of the underlying microarchitecture). Hence, any procedure for detecting TAs based on those definitions should be exact, excluding false positives/negatives. Our assessment shows that *no definition* is able to identify TAs precisely on all the considered examples.

CONTENTS

5 Interpretation and Modeling of the Definitions . . . 85
5.1 Formal Modeling of the OoO-Pipeline Template . . . 85
5.1.1 Abstract Modeling for Timing Properties . . . 85
5.1.2 Formal Specification . . . 86
5.2 Assumptions on the Definitions . . . 91
5.2.1 Step Heights in Step Functions . . . 92
5.2.2 Intersections in Step Functions . . . 93
5.2.3 Component Occupation . . . 93
5.2.4 Instruction Locality . . . 94
5.3 Uniform Formal Modeling of Properties . . . 96
5.3.1 Discussion on Hyperproperties . . . 96
5.3.2 Properties Based on the Definitions . . . 96
5.4 Summary: the Parameters of our Formal Model . . . 101
6 Assessment of the Definitions . . . 103
6.1 Assessment by Model Checking . . . 103
6.1.1 Verification Methodology . . . 103
6.1.2 Shortcomings of the Definitions . . . 104
6.2 Assessment Outcome . . . 110
6.2.1 Unsuited Granularities for Detecting TAs . . . 110
6.2.2 Towards the Notion of Causality . . . 111
6.3 Summary: the Lack of Causality . . . 112

5 – INTERPRETATION AND MODELING OF THE DEFINITIONS

In this chapter, we present our formal framework to evaluate the presence or absence of TAs under the various definitions, in a unified, pairwise interpretation. We aim at making this assessment by model checking, over executions of specific traces on the representative OoO microarchitecture introduced in Ch. 2.

First of all, we need a *formal* model of the OoO-pipeline case study introduced in Sec. 2.3. We thus first introduce our parametric formal model of the considered microarchitecture (Sec. 5.1). We cannot encode procedures as is, since none of the existing definitions appears to be precise enough to be systematically used for detecting TAs in a concrete microarchitecture—the definitions are essentially theoretical and only exemplified through simple scheduling diagrams in functional units. Thus, we have then to specify precise assumptions so as to make all definitions applicable to a systematic detection of TAs in our case study (Sec. 5.2). Finally, we can formulate predicates based on the formal model and these assumptions, as detection procedures related to the various definitions (Sec. 5.3).

5.1 . Formal Modeling of the OoO-Pipeline Template

In this section, we present our *formal* and *executable* TLA^+ [48] model of the OoO-microarchitecture template introduced in Sec. 2.3. This model targets mainly the non-functional, *timing* behavior of the pipeline, described in Sec. 2.3.2, for the purpose of verifying timing properties, in particular those related to TAs. We detail the general modeling needs in this purpose (Sec. 5.1.1), before providing an insight of the formal specification of the OoO pipeline (Sec. 5.1.2).

5.1.1 . Abstract Modeling for Timing Properties

The formal modeling of OoO microarchitectures generally focuses exclusively on *functional* correctness, often from specialized decision procedures, in particular to verify implementations of Tomasulo's algorithm (cf. Sec. 3.4). To our knowledge, few formal models of OoO hardware intended for the model checking of timing properties have been developed. We mentioned several model-checking approaches in Sec. 4.4.1, in particular by Dalsgaard et al. [93] and Gustavsson et al. [94], which target simple *in-order* microarchitectures. These approaches model the advancement of instructions through the in-order pipeline stages, in a similar way to our formal modeling of the OoO template; nevertheless, they rely on specific time abstractions (clocks [65]) that we do not need to express our timing properties and to verify them through generic procedures (cf. Sec. 3.3). Eisinger et al. [79] model an OoO pipeline but they do not provide modeling details (cf.

Sec. 4.4.1). The formal specification introduced in this section is inspired by the models by Asavoae et al. [34, 96], nevertheless with a more generic, parametric approach and with the modeling of a reorder buffer (ROB).

A suitable model for verifying timing properties necessarily integrates both hardware and software features, whose combination characterizes the system and in particular its non-functional timing characteristics. The properties are not correlated to the functional complexity of microarchitectures, which materializes into the data path (cf. Sec. 1.1.1). **We do not need to consider the functional aspects beyond their impact on the pipeline-level timing behavior.** Instead, we need to develop an abstract formal model of the processor focusing on the instruction progress—the software characteristics—through the successive pipeline stages of the processor—the hardware characteristics. The data path is abstracted into black-box, but cycle-accurate pipeline stages imposing timing constraints to the instruction flow, except for the execute stage, where in addition the scheduling algorithm that maps instructions to functional units is fully captured. Finer models are unnecessary since they describe changes in internal, hidden states of the data path, typically a matter of functional correctness. On the contrary, pipeline-level models are required, since pipeline stages are essential to the cycle-accurate timing behavior, which allows observing external events, e.g., the full completion of an instruction.

Our abstraction thus needs to precisely delimit the pipeline stages from the data path of the hardware microarchitecture, and to extract the control signals that impact the timing behavior of the control path according to the pipeline stalling logic. We also need to map at any time instructions onto the identified stages that process them, from the input program—this is the combination of the hardware and software specifications. Finally, our model also captures the execution time to verify timing properties.

5.1.2 . Formal Specification

Hereafter, we exemplify the main features of a suitable formal model for the verification of timing-predictability properties, from the representative OoO-pipeline template introduced in Sec. 2.3. We present the key points for modeling the timing behavior of Tomasulo's algorithm with a ROB, for the purpose of verifying properties that indicate the absence of TAs under various definitions (Sec. 5.3).

We formalized this pipeline using the TLA^+ language [48] (cf. Sec. 3.3.2).[1] In our case, **the transition system defines how the instructions of an execution trace proceed through the pipeline at the granularity of clock cycles**. We model which instruction is processed by each of the hardware components depicted

[1] In this chapter, we present a simplified version of the model, where the data dependencies are explicitly stated in the input $program$ while the ROB is implicit (it only stores instruction indexes), and where the ROB and the RSs are infinite buffers. The related sources are available at: `https://bitbucket.org/benjaminbinder/ta-models/` in branch `master`. A refined version of the specification is used in Part III.

in Fig. 2.3, at any instant. In our abstraction, the pipeline stages do not have side effects, such as a write to the memory or the register file. We consider multi-cycle instructions that thus may cause stalling. The pipeline timing behavior depends on the number of units for each stage (cf. $superscal$ and N_{FU} in Fig. 2.3), on the program dependencies that clearly restrict OoO computations, and, when needed, on the mere information of the required computation clock cycles.

We systematically verified that our specification behaves as intended, namely as described in Sec. 2.3.2. To do so, we relied on simple, specific invariants, e.g., when we specified the scheduling on FUs, an invariant for verifying that data dependencies are always respected. We will also represent concrete counterexamples showing detailed execution scenarios in Ch. 6.

Abstract Data Path and Computations

We define a state variable for each pipeline stage ($\_IF$, $\_ID$, $\_RS$, $\_FU$ and $\_COM$), which notably contains the instructions that are currently processed. The specification depends on a set of input parameters (a in Table 5.1, Sec. 5.4) that enable to set the instruction sequence under analysis ($program$) and refine the characteristics of the microarchitecture ($superscal$ and N_{FU}). The last two parameters represent a particular version of the pipeline template, by fixing its abstract data path. Execution parameter $program$ specifies the input instruction sequence with increasing addresses[2] associated with execution constraints about the mapping onto the hardware. This information originates from the analysis of the concrete program: each instruction embeds the set of admissible functional units (FU affinities)—an abstraction of the functional instruction class—, as well as a set of possible *latencies* related to timing-variable stages—an abstraction of the intended computations. The variables related to such stages contain, besides the current instruction, the elapsed latency (i.e., a counter) and the total required latency in the stage. Memory is not explicitly modeled, but the IF stage and the FUs feature instead a variable timing behavior. Execution constraints thus specify latencies in the IF stage (modeling instruction cache hits/misses)[3] and in the FUs (e.g., modeling the data-cache behavior). The *actual* total latencies in these stages result from arbitrary choices among the sets of possible latencies (in $program$).

The register file is not modeled either, but only the (Read-After-Write) data dependencies ($\downarrow$), which are explicitly encoded in $program$. **The resulting abstract specification allows for all the behaviors that are concretely made possible by different initial hardware states** (e.g., the initial cache content), considering the execution of the input instruction sequence on the target microarchitecture. It remains to actually make instruction classes progress through the pipeline, i.e., to encode the control path from the established data path and the

[2]We exclude branch instructions, thus focusing on one program path.

[3]In our model, an instruction cache miss in one superscalar IF stage stalls all IF stages.

execution constraints.

Timing Modeling of the Control Path

While the data path refers to the hardware *states*, the control path materializes the *transitions* that entail changes in the data-path configuration. We consider that the pipeline is initially empty. In order to get a cycle-accurate abstraction of the control path, a transition models *one* clock cycle, where the change in the state of each stage is modeled by an action involving a data-path variable—this constitutes the body of operator $Progress$ involved in the next-state relation introduced in Fig. 3.1. The additional state variable $currCycle$ is a counter modeling absolute time (cf. Sec. 3.3.2): $currCycle' = currCycle + 1$ holds as long as there is a change in the state of a pipeline stage (see Fig. 3.1), i.e., the input sequence has not fully executed. The evolution of absolute time is modeled in accordance by operator $TimeProgress$, which is used as a conjunct in the next-state relation (Fig. 3.1). Finally, state variable $prog$ is a record monitoring the execution, with a field ($rest$) containing the remaining instructions (not fetched yet) from $program$ and a field ($exec$) modeling the ROB. Each of these fields are sequences of instructions, where the instructions are nested records, e.g., with Booleans $completed$ and $committed$ for the instruction status in the ROB.

We now illustrate how the abstract data-path state is used in order to accurately model the control path, by *focusing on the most critical, OoO-specific elements of the control path*, namely the scheduling of instructions to FUs and the reordering for in-order commit. Modeling the scheduling requires selecting the next pending instruction and modeling the in-order commit, the next completed instruction. Both selections rely on the instruction status in the ROB.

Prerequisite Operators. First of all, let us define operators $Exec$ and $Done$, used to specify the next-state relation. They return the set of the ROB indexes of the instructions that will have already completed and, respectively, committed, in the next cycle.

$$NxtFUBusy(i) \triangleq \_FU[i].currLat < \_FU[i].baseLat \quad 1$$
$$Exec \triangleq \{i \in 1 \,..\, Len(prog.exec) : \quad 2$$
$$\lor\ prog.exec[i].completed \quad 3$$
$$\lor\ \exists j \in 1 \,..\, N_{FU} : prog.exec[i].PC = \_FU[j].PC \land \neg NxtFUBusy(j)\} \quad 4$$
$$Done \triangleq \{i \in 1 \,..\, Len(prog.exec) : \quad 5$$
$$\lor\ prog.exec[i].committed \quad 6$$
$$\lor\ \exists j \in 1 \,..\, superscal : prog.exec[i].PC = \_COM[j].PC\} \quad 7$$

Operator $Exec$ returns the set of indexes in the range of the current ROB (line 2, first line of the definition of $Exec$) s.t. the relevant Boolean field ($completed$) of the corresponding instructions ($exec[i]$) is set (first disjunct, line 3) or a back-to-back execution, i.e., without delay (see -·▸ in Fig. 2.3), is possible (second disjunct, line 4). In the latter case, the instruction itself (PC field, whose values are unique across program instructions) is currently handled by one of the FUs,

i.e., it is the instruction of the j-th element of the $\_FU$ state variable (first conjunct in line 4), and the instruction in this FU is to leave the FU in the next cycle (second conjunct in line 4). Indeed, operator $NxtFUBusy(i)$ uses the information about latencies contained in the $\_FU$ variable to determine whether the instruction currently handled by a given FU should remain in the FU in the next cycle and, hence, cause a pipeline stalling. This operator compares the current latency $currLat$ of the i-th FU with the total required latency $baseLat$.

Similarly, operator $Done$ is based on the relevant field of the ROB ($committed$), as well as on the ongoing commits ($\_COM$ state variable) that will be immediately echoed to the ROB. Note that the commit is always a one-cycle operation in our model. Operator $Exec$ and $Done$ are both used to update the ROB field $exec$ of $prog$ in each cycle.

Scheduling on the FUs. Based on operator $Exec$, we can now specify the scheduling of instructions to the FUs. Operator $NxtFU(i)$ returns the instruction that is to be scheduled to the i-th FU in the next cycle, or a special instruction $empty$ that models the absence of an instruction:

$$
\begin{array}{lll}
FURout(i) \triangleq \{\_ID[j].PC : j \in \{k \in 1 \,..\, superscal : & & 1\\
\qquad \_ID[k].PC \neq empty \wedge FU[\_ID[k].PC.pc] = i\} & & 2\\
NxtFU(i) \triangleq \text{IF } NxtFUBusy(i) \text{ THEN } empty & & 3\\
\qquad \text{ELSE LET } minReady \triangleq Min(\{x.pc : x \in & & 4\\
\qquad \{y \in \_RS[i] \cup FURout(i) : \forall\, z \in y.dep : z \in Exec\}) \text{ IN} & & 5\\
\qquad \text{IF } minReady = 0 \text{ THEN } empty & & 6\\
\qquad \text{ELSE CHOOSE } x \in \_RS[i] \cup FURout(i) : x.pc = minReady & & 7\\
 & & 8
\end{array}
$$

In the case that the i-th FU does not suffer stalling and thus may accept a new instruction in the next cycle (i.e., $NxtFUBusy(i)$ evaluates false), we define local operator $minReady$ (lines 4-5) that determines the *address* (pc field from the $program$ input parameter) of the relevant instruction among the candidate instructions. If this instruction exists (0 is the conventional address of the $empty$ instruction, used in line 6), we select (through the TLA^+ CHOOSE operator) the *instruction* itself whose address has been determined by local operator $minReady$ (line 7). $minReady$ implements an age-ordered policy that selects the oldest instruction whose all dependencies are satisfied (or will be in the next cycle). It is based on the assumption that older, preceding instructions in the program order, have smaller addresses (see above). It is also based on operator $FURout(i)$ providing the set of the currently decoded instructions (in the ID stage) that have *actually* been assigned the FU under consideration, which requires the FU to be admissible for this instruction. This is trivial when the decoded instructions have only one admissible FU and it otherwise lies on an arbitrary choice. Consequently, $minReady$ selects the smallest address (Min), from the instructions waiting in the

associated RS[4] or directly from the ID stage[5], more precisely only those (set in line 5) whose all dependencies (dep field assigned from the $program$ input parameter) will have been computed (i.e., in $Exec$).

The issued instructions are removed from the RSs in accordance, while the non-issued decoded instructions are added for a later selection, the whole through simple set-theory operators. Each entry of the $\_RS$ variable (one per RS/FU) is updated under this consideration:

$$\_RS' = [i \in (1 \,..\, N\_FU) \mapsto (\_RS[i] \cup FURout(i)) \setminus \{NxtFU(i)\}]$$

Similarly, the $\_FU$ variable is updated using the $NxtFU(i)$ operator for each FU i.

In-Order Commit. Hereafter, we focus on the second main item of the control path, namely the way of exploiting the data path so as to specify the in-order commit of instructions. An instruction can be committed in the next cycle only if its computation in a FU has completed (in the next cycle) and all the preceding instructions in the ROB sequence have committed. Note that operator $Done$ does not suffice to consider all the preceding committed instructions, since several commits might be performed in the next cycle, depending on $superscal$. We define operator $ReadyCOM(DoneCOM)$, returning the set of instructions that can be committed:

$ReadyCOM(DoneCOM) \triangleq$ 1
LET $ready \triangleq \{j \in 1 \,..\, Len(prog.exec) : \forall\, k \in 1 \,..\, j-1 : k \in DoneCOM\}$ 2
IN $(Exec \cap ready) \setminus DoneCOM$ 3

For simplification, we consider the scalar case (i.e., $superscal = 1$) in the first instance, in which parameter $DoneCOM$ is always equal to the set derived from operator $Done$. Local operator $ready$ returns the set of instruction indexes whose all older instructions will have been committed in the next cycle (i.e., in $DoneCOM$). This set is intersected with that of the indexes of completed instructions, determined by operator $Exec$. Instructions are never actually removed from the ROB, in order to keep information about the whole execution of the considered behavior. This will be useful to specify and verify properties. Hence, the already committed instructions must be removed from the intersection of the candidate instructions. Finally, the (potential) instruction that commit in the next cycle is the oldest instruction among those that are ready for commit, determined from the minimum of the resulting set (in the same way as in $NxtFU(i)$).

In the superscalar case ($superscal > 1$), $ReadyCOM(DoneCOM)$ must be applied $superscal$ times in the same transition, taking into account that a first committed instruction might immediately make another instruction committable

[4]This models the forwarding through the Common Data Bus (CDB), which broadcasts the results from the FUs (see Sec. 2.3.1).

[5]A decoded instruction is immediately issued to the FU if it is ready to execute and the related RS is empty (see Sec. 2.3.2 and ⋯► in Fig. 2.3).

(if the latter instruction has also already executed and is only waiting in the ROB for the former to commit). This may provoke a chain reaction of simultaneous commits that is only limited by the $superscal$ number of resources. Then, the (at most) $superscal$ instructions that will actually commit in the next cycle result from the $superscal$ sets derived from $ReadyCOM(DoneCOM)$:

$$\begin{array}{ll} \text{RECURSIVE } nxtCOM(\_) & 1 \\ nxtCOM(s) \triangleq \text{ IF } s = 1 \text{ THEN } ReadyCOM(Done) & 2 \\ \quad \text{ELSE } ReadyCOM(Done \cup \text{UNION } (\{nxtCOM(j) : j \in 1 \,..\, s-1\})) & 3 \end{array}$$

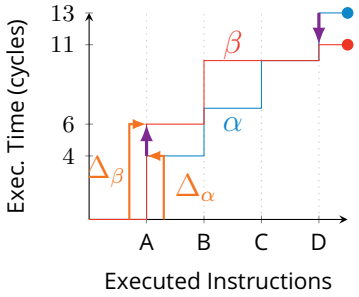
These sets are constructed recursively (line 1) through operator $nxtCOM(s)$, called $superscal$ times, where the base case (line 2) corresponds to a scalar pipeline, i.e., $ReadyCOM(Done)$, whereas in the recursive steps (line 3), the value of parameter $DoneCOM$ is a set that also contains (the indexes of) all the preceding instructions that have just been determined to commit in the next cycle.

Note that operator $ReadyCOM(DoneCOM)$ does not remove from the intersection the set given by $Done$, but indeed by $DoneCOM$. Thus, a call to $ReadyCOM(DoneCOM)$ can never provide a set that contains an instruction index in $DoneCOM$ and, as a consequence, $nxtCOM(s)$ excludes the instructions that have already committed (i.e., instructions are not committed twice). Besides, a closer look reveals that local operator $ready$ (within $ReadyCOM(DoneCOM)$) is such that for a given value of $DoneCOM$, the resulting set is either the empty set or a singleton, since if in the next cycle, an instruction has executed but not committed yet, either none or *one* instruction only will have all preceding instructions committed[6]. Consequently, each call to $nxtCOM(s)$ brings at most one new instruction to $DoneCOM$ in the next recursive call ($s+1$), which can be selected as the instruction to commit for this value of $s+1$. Since, moreover, the operator is called at least once per cycle (depending on $superscal$), no instruction commit may be missed.

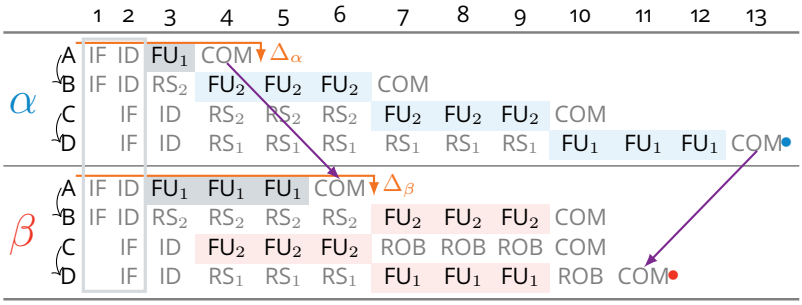
5.2 . Assumptions on the Definitions for the Concrete OoO-Pipeline Model

In order to make all definitions applicable and evaluate their capabilities in the detection of TAs, we need to rely on a set of interpretations/assumptions (which are to be detailed and explained when needed). We address the problems highlighted in Sec. 4.2, for our hardware model, by revisiting the definitions. We generally focus on *pairs* of traces only—this is not restrictive in our pairwise interpretation (cf. Sec. 4.1). We illustrate our assumptions from the common example in Fig. 2.4, which results, with specific graphical representations, in Fig. 5.1.

[6] Assume that the resulting set contains a second instruction and, without restricting the generality, that this instruction follows in the program. Necessarily, all preceding instructions in the program have committed or are in COM, thus in particular the first considered instruction, which is contradictory.



(a) Execution time as step function.



(b) Execution traces showing latencies ( ), the order of commits ( ), the assignment to functional units (/ /), the common prefix ( ) considered to apply the definition based on locality, and the end of traces (/).

Figure 5.1: Different ways of representing two execution traces that constitute a counter-intuitive TA on our OoO-pipeline model (with $superscal = 2$ and $N_{\mathsf{FU}} = 2$, cf. Sec. 2.3), from a given program (e.g., 2.4a) showing data dependencies ( ).

5.2.1 . Step Heights in Step Functions

The evolution of the cumulative execution time of a trace can be represented as a *step function*. Fig. 5.1a represents this step function for the two traces in Fig. 5.1b. This plot is obtained from the cumulative execution times up to the end of the execution of each instruction in both traces, i.e., α that we assume to start from initial state η_α and β that we assume to start from η_β:

$$\begin{cases} \Gamma(\eta_\alpha, A) = 4 & \Gamma(\eta_\beta, A) = 6 \\ \Gamma(\eta_\alpha, B) = 7 & \Gamma(\eta_\beta, B) = 10 \\ \Gamma(\eta_\alpha, C) = 10 & \Gamma(\eta_\beta, C) = 10 \\ \Gamma(\eta_\alpha, D) = 13 & \Gamma(\eta_\beta, D) = 11 \end{cases} \tag{5.1}$$

The main issue with this class of formal definitions concerns the definition of *latencies* (cf. Sec. 4.2.1). For this approach, **we thus assume instruction latencies as the number of cycles between the commit of an instruction (COM) and the previous commit (or trace start)**, according to the program order of instructions. The instruction latencies in both traces in Fig. 5.1a are thus

the following:

$$\begin{cases} \gamma(\eta_\alpha, 1) = 4 & \gamma(\eta_\beta, 1) = 6 \\ \gamma(\eta_\alpha, 2) = 3 & \gamma(\eta_\beta, 2) = 4 \\ \gamma(\eta_\alpha, 3) = 3 & \gamma(\eta_\beta, 3) = 0 \\ \gamma(\eta_\alpha, 4) = 3 & \gamma(\eta_\beta, 4) = 1 \end{cases} \tag{5.2}$$

Note that it indeed holds that for $\eta \in \{\eta_\alpha, \eta_\beta\}$, for all $i \in \{A, B, C, D\}$, $\gamma(\eta, i) \geq 0$ and for all $n > i$ (in lexicographical order), $\Gamma(\eta, n) = \sum_{i=1}^{n} \gamma(\eta, i)$.

Then, a TA can be identified by comparing the step functions of two traces: a TA occurs when the latency for an instruction in one trace, i.e., the corresponding step height in the plot, is smaller than in the other trace, but later the execution time is larger. This is illustrated by Fig. 5.1a, where the step height Δ_α ($\gamma(\eta_\alpha, A) = 4$) of trace α is smaller than Δ_β ($\gamma(\eta_\beta, A) = 6$) of trace β, but the execution time of α at the end of the trace ($\bullet\ \Gamma(\eta_\alpha, D) = 13$) is larger than that of β ($\bullet\ \Gamma(\eta_\beta, D) = 11$). For brevity, the step functions can be observed directly in the detailed trace representation of Fig. 5.1b, where the steps are represented in the same way.

5.2.2 . Intersections in Step Functions

In the pairwise interpretation, all definitions of this class apply in the same manner, which can again rely on step functions. A TA occurs when the step functions of both traces *intersect*, i.e., a trace that initially executed instructions *faster* suddenly becomes *slower* than the other. This class of definitions only relies on the cumulative execution time, which can be determined unambiguously; hence, it does not require specific interpretations.

This situation is illustrated in Fig. 5.1a, where the traces intersect at the last instruction: α initially situated below β passes above. From the existence of such an intersection, it follows that the absolute values of the step functions switch order, as indicated by the red arrows (↑ and ↓).

Such an inversion can also be observed in the detailed execution traces from Fig. 5.1b, by looking at the respective instances at which instructions were committed (COM). Instruction A, for instance, was committed in cycle 4 for α but in cycle 6 for β, as illustrated by the red diagonal arrow. The situation is inverted for instruction D, as indicated by the red arrow pointing in the *opposite* direction. Red arrows pointing in opposite directions (↘ vs. ↖) then indicate a TA (similarly to intersections in plots).

5.2.3 . Component Occupation

The main problem with this definition resides in the fact that suitable partitions for considering hardware components are not defined (cf. Sec. 4.2.3). We do not need to explore all possible partitions to assess the definition. **We assume any non-empty subset of the FUs to be a possible component.** In our formal model, the considered subset is supplied as an input parameter and we assume that

a TA exists if a parallel inversion is signaled for the FU(s) of this subset.

5.2.4 . Instruction Locality

The definition based on locality is the most complicated to apply. When we try to apply it in the pairwise interpretation, the main issue concerns locality constraints (cf. Sec. 4.2.4). Since this definition is strongly related to WCET analysis, we also illustrate again, henceforth from detailed traces, the limitations of the absolute-WCET interpretation.

Locality Constraints. This definition relies on the decomposition of traces from the transition system that describes the cycle-accurate behavior of the pipeline. For this work, we assume that the transition system can be represented by a table, similar to the one from Fig. 5.1b, that assigns instructions to pipeline stages. We assume that **every instruction occupies only one location at a time** in a trace (cf. "convex predicates" in Sec. 4.2.4). Since the RSs and the ROB only model instructions waiting for a FU or for committing, i.e., due to the scheduling of other instructions, we furthermore assume **all pipeline stages except the RSs and the ROB to be part of the set of suitable locations**. We also assume that **FUs are gathered in a single execution stage (EX) as a suitable location**. This allows (local) comparisons of the occupation even when different FUs are used for the execution of an instruction. Finally, we do not compare the occupation of locations after the moment when the traces diverge.

Consider again Fig. 5.1b, in order to illustrate the situation. We denote the hardware state in cycle i as α_i for trace α and β_i for β, so that $\alpha = (\alpha_1, \ldots, \alpha_{13})$ and $\beta = (\beta_1, \ldots, \beta_{11})$. We denote the locality constraint indicating that instruction X occupies stage S as $S(X)$ (which is true of a state if X is in S in this state). The depicted traces are clearly identical for the first two cycles, with the common prefix $\alpha_{\text{pre}} = (\alpha_1, \alpha_2) = (\beta_1, \beta_2)$, as indicated in the figure by the gray box on the left ( ). We can decompose α and β in the following way (cf. Sec. 4.2.4): $\alpha = \alpha_{\text{pre}} \circ \alpha_{|l} \circ \alpha_{post}$ and $\beta = \alpha_{\text{pre}} \circ \beta_{|l} \circ \beta_{post}$. This decomposition is possible only with the following locality constraints (according to the above): $l = \text{ID}(C)$, $l = \text{ID}(D)$, or $l = \text{EX}(A)$ (see the column of the third cycle in Fig. 5.1b). We can then compare the sub-traces in which each of these locality constraints holds in both traces. Locality constraints $\text{ID}(C)$ and $l = \text{ID}(D)$ hold in α_3 and β_3 only:

$$\left|\alpha_{|l=\text{ID}(C)}\right| = \left|\alpha_{|l=\text{ID}(D)}\right| = \left|\beta_{|l=\text{ID}(C)}\right| = \left|\beta_{|l=\text{ID}(D)}\right| = 1 \tag{5.3}$$

However, a non-deterministic choice in cycle 3 causes the traces to diverge. As a result, FU_1 is occupied by 1 vs. 3 cycles for α and β, respectively ( ):

$$\alpha_{|l=\text{EX}(A)} = (\alpha_3) \qquad \beta_{|l=\text{EX}(A)} = (\beta_3, \beta_4, \beta_5) \tag{5.4}$$

It holds, for all the above locality constraints, that: $\beta_{|l} \geq \alpha_{|l}$, thus β is *local-worst-case* (and, since we face two traces only, α is non-local-worst-case). The

TA is explained by the fact that β is the local-worst-case trace, while yielding a shorter execution time than non-local-worst-case trace α (see again the red arrows in Fig. 5.1b)—actually since instruction A occupies FU_1 longer in this trace, which has an impact on the subsequent instruction scheduling.

Comments on the absolute-WCET interpretation. We illustrate the limitations of the absolute-WCET interpretation from our concrete OoO-pipeline template, hence in a hardware-centric interpretation, though the original definition is analysis-centric (cf. Sec. 4.1)—it assumes a hardware abstraction, and consequently, a static analysis to compute abstract hardware states. However, on the one hand, the definition does not indicate suitable abstractions, and, on the other hand, we can apply the definition to an abstract model arbitrarily close to our concrete model, without loss of generality. We thus intend to illustrate the situation schematized in Fig. 4.4b, which might be derived from an abstract model in any case (cf. Sec. 4.1). We assume an abstraction that explores all (concrete) states and we consider that an abstract state maps each pipeline resource to the (potential) processed instruction at a given instant and to the associated latency.

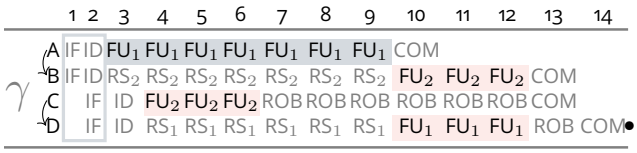


Figure 5.2: A third execution trace (γ), in addition to those in Fig. 5.1b. All traces exhibit a local variation wrt. the use of FU_1 by instruction A ( ). The three traces constitute the situation schematized in Fig. 4.4b, where γ is the local-worst-case trace and yields the WCET, so that no TA is stated in the absolute-WCET interpretation, which justifies that we focus on the pairwise interpretation.

Let us consider, from Fig. 5.1b, a third trace, γ, represented in Fig. 5.2. Consequently, instruction A experiences a trivalued variation, e.g., a data cache hit that is performed, in certain cases, with a write-back to the main memory, entailing an additional penalty. Instruction A executes 1 cycle on FU_1 in α (hit), 3 cycles in β (miss), and 7 in γ (miss with write-back). The three traces share the same prefix in the first two cycles, so that the definition prompts to state γ as the local-worst-case trace, through the use of FU_1 by A. Since γ is also the longest trace, i.e., yielding the WCET, the definition states the absence of TAs. This constitutes exactly the situation represented in Fig. 4.4b. If, in certain cases, a write-back to the main memory is impossible, the initial state of γ, and thus this trace, should be excluded (i.e., for a tighter WCET bound). Yet, the TA that concretely occurs between α and β is not captured by the definition as it is. This is the reason why we focus on the pairwise interpretation (cf. Sec. 4.1).

5.3 . Uniform Formal Modeling of Properties

The parameterized formal specification of the pipeline template (cf. Sec. 5.1.2) allows us to model the execution of a single (arbitrary) execution trace, while tracking all the instructions of the trace, as well as all the involved hardware components. However, in order to reason about TAs, we need at least two traces—exactly two in our pairwise interpretation (Sec. 5.3.1). This formal framework allows us to implement the various definitions of TAs into a uniform formalization (Sec. 5.3.2).

5.3.1 . Discussion on Hyperproperties

A TA is actually a (safety) *hyperproperty* [103] of the *pipeline* specification, namely a safety property (cf. Sec. 3.2.1) that is a set *of sets* of traces, instead of a set of traces for Linear-Time (LT) properties (Def. 3.12)—more specifically a *2-safety* property, since it is a set of pairs of traces. To reason about two traces simultaneously, we define a self-composition [104] of the pipeline specification, through two instances in a main, higher-order specification module. **All state variables are duplicated and each instance manipulates its own set.** In this way, we can formulate a common (LT) *safety* property of the *dual execution* in the main module, instead of a 2-safety property of the basic pipeline specification.[7] Both instances share the input parameters (cf. a in Table 5.1), which guarantees that we consider the same program and the same version of the data path of the microarchitecture.

We thus face two *essentially* identical copies of the microarchitecture model. The traces are thus restricted to the same instruction sequence, having the same set of dependencies, possible latencies and FU affinities, on the exact same hardware microarchitecture. The differences between the two traces arise solely from variations in the *actual* latencies and FUs of instructions observed during execution (see Sec. 5.1.2). Instructions may advance at their own pace through the pipeline, according to their own actual latencies and assignments to FUs, the use of hardware components by other instructions, and the dependencies among instructions.

5.3.2 . Properties Based on the Definitions

The uniform formalization consists in defining, in association with each definition, **a detection procedure that decides whether a TA is signaled for the considered definition**. These procedures are specified in the form of *invariants* (cf. Sec. 3.2.1) drawing on elements of the pipeline model. In this regard, we rely on additional code and state variables, summarized in Table 5.1. The table highlights helper operators (b), which operate on the history of a trace, and helper state variables (c).

We may observe a TA only when *both* executions have completed, at least up

[7] Besides, note that hyperproperties are not supported by TLC.

to a certain instruction. The invariants, explained in detail further below, check the absence of TAs after the completion of each instruction in both traces, as a direct formalization of the key ideas and the assumptions introduced in Sec. 5.2. Operator $ProgDone(n)$ (cf. Table 5.1) returns a Boolean indicating whether both executions have completed (at least) up to the n-th instruction of the input sequence:

$$ProgDone(n) \triangleq \forall\, tr \in 1\,..\,2 : \exists\, i \in 1\,..\,Len(progs[tr].exec) : \quad 1$$
$$progs[tr].exec[i].PC.pc = Program[n].pc \wedge progs[tr].exec[i].committed \quad 2$$

$ProgDone(n)$ checks whether the n-th instruction exists in the ROB (first conjunct line 2) of the instances corresponding to both traces ($tr \in 1\,..\,2$)—*progs* is an operator returning a pair containing both copies of state variable *prog* (cf. Sec. 5.1.2)—and whether its execution is over (*committed* field, second conjunct line 2).

Property for Step Heights in Step Functions

The key elements of this definition are commit events (cf. Sec. 5.2.1), which are tracked by an additional field in state variables and accessible via helper operator $ComTime(tr, n)$ (cf. Table 5.1). This additional field *comTime* nested in the ROB entries (cf. Sec. 5.1.2) keeps track of the instant (*currCycle*) of each commit event occurring during the execution. Operator $ComTime(tr, n)$ returns the value of the *comTime* field for the n-th instruction in the trace specified through parameter $tr \in \{1, 2\}$:

$$ComTime(tr,\, n) \triangleq progs[tr].exec[n].comTime$$

A TA thus occurs when the comparison of the k-th instruction's *step heights* does not match the global execution time—or, inversely, a TA is excluded when they always match, as expressed here. We use operator $StepHeight(tr, n)$, which is in fact derived from the commit time (cf. Table 5.1). Based on the operators, we now specify the property expressing the absence of TAs under this definition:[8]

$$NoTASteps \triangleq \forall\, k \in 1\,..\,Len(Program) - 1 : \forall\, n \in k + 1\,..\,Len(Program) : \quad 1$$
$$\wedge\, ProgDone(n) \quad 2$$
$$\wedge\, StepHeight(1,\, k) < StepHeight(2,\, k) \quad 3$$
$$\Longrightarrow\; ComTime(1,\, n) \leq ComTime(2,\, n) \quad 4$$

There is no TA iff, for any instruction k and for any *subsequent* instruction n (line 1), it holds that if:

1. the execution is completed up to the considered instructions in both instances under consideration (line 2),
2. and the (local) commit for instruction k is s.t. the step height (line 3) is smaller in the first instance (α in Fig. 5.1) than in the second one (β),

then the commit ordering for subsequent instruction n in both instances is the same as that of the step heights (line 4).

[8] Implication has the lowest precedence (cf. Sec. 3.3.2).

Note that both instances of the pipeline specification are totally interchangeable. That justifies the fact that we fix a priori the roles of each trace in the property, namely their commit ordering—it is not necessary to consider the case where both traces switch positions in the formula, since TLC will explore all possible pairs of traces. This also applies to the subsequent formulae.

Property for Intersections in Step Functions

The predicate for this definition (Sec. 5.2.2) is very similar as the previous one. Here, a TA occurs when the intermittent order of the k-th instruction's commit (itself, instead of the step height) between the two traces (second conjunct) does not match the global execution time:

$$
\begin{aligned}
NoTAInter \triangleq\ & \forall\, k \in 1\,..\,Len(Program) - 1 : \forall\, n \in k + 1\,..\,Len(Program) : && 1\\
& \land\ ProgDone(n) && 2\\
& \land\ ComTime(1,\, k) < ComTime(2,\, k) && 3\\
& \implies\ ComTime(1,\, n) \leq ComTime(2,\, n) && 4
\end{aligned}
$$

Property for Component Occupation

In order to express the predicate based on component occupation (Sec. 5.2.3), additional state variables have to be added to the TLA^+ specification that track the occupation of FUs, which is again accessible through a helper operator, $FUuse(tr, f)$ (cf. Table 5.1):

$$
\begin{aligned}
NoTAComp \triangleq\ & \text{LET } n \triangleq Len(Program) \text{ IN} && 1\\
& \quad \text{LET } use(tr) \triangleq \textstyle\sum_{f \,\in\, locFU} FUuse(tr,\, f) \text{ IN} && 2\\
& \quad \land\ ProgDone(n) && 3\\
& \quad \land\ use(1) < use(2) && 4\\
& \quad \implies\ ComTime(1,\, n) \leq ComTime(2,\, n) && 5
\end{aligned}
$$

From the individual occupation obtained through $FUuse(tr, f)$ (not detailed), the occupation of the supplied component is computed through summation.[9] The FUs to consider in this component are provided as a model parameter, $locFU$ (cf. Table 5.1), since the choice of the partitions has little impact on the evaluation presented in Ch. 6. As before, the absence of TAs is stated when the relationship between the component occupation (use at the second conjunction) of the two traces is always the same as the global execution time (the consequent in the implication).

Property for Instruction Locality

The invariant for this definition (Sec. 5.2.4) is the most complex, since it is not based on simple numeric features as the other definitions. Firstly, TAs are associated with the instant when the two traces *diverge*. As explained in Sec. 5.2.4, we consider the two traces identical as long as the mapping of instructions to pipeline

[9] This is a pseudo-TLA formula, where only the formalization of the summation is simplified.

stages is identical. This is expressed through state variable $commonPre$ (cf. Table 5.1), which is initialized to TRUE and only reset to FALSE when this mapping differs:

$$
\begin{aligned}
commonPre' = \ & \wedge commonPre && 1\\
& \wedge \forall\, tr \in 1\,..\,2 : \forall\, k \in 1\,..\,superscal : \exists\, kk \in 1\,..\,superscal : && 2\\
& \quad \text{Treatment for the IF stage} && 3\\
& \quad \wedge IFs[tr][k].PC = IFs[3-tr][kk].PC && 4\\
& \quad \wedge IFs[tr][k]'.PC = IFs[tr][k].PC && 5\\
& \qquad \Longrightarrow IFs[3-tr][kk]'.PC = IFs[3-tr][kk].PC && 6\\
& \wedge \forall\, tr \in 1\,..\,2 : \forall\, k \in 1\,..\,N_{FU} : \exists\, kk \in 1\,..\,N_{FU} : && 7\\
& \quad \text{Similar treatment for the EX stage (gathering FUs)} && 8
\end{aligned}
$$

In our model, this is expressed through the terms $IFs[tr][k]$ (and $FUs[tr][k]$), where operators IFs (and FUs) are pairs—one entry per execution trace, similar to $progs$ above. Each pair contains a sequence tracking the content of any IF stage (and FU, respectively) k in trace tr, from both copies of state variables $\_IF$ (and $\_FU$). These terms give access to the assignment of instructions to the IF stages (there are several if $superscal > 1$) (and the FUs) (fields PC), and also to the current (elapsed) latency of the instruction in the stage (or FU) (field $currLat$). Any divergence in a PC field causes $commonPre$ to be reset. The first main-level conjunct (line 1) ensures the whole conjunction to *remain* FALSE once both traces have diverged. The second disjunct (lines 2-6) ensures the conjunction to evaluate to FALSE when the traces diverge in the next cycle due to IF, which does *not* occur when the following two conditions hold for any trace tr, i.e., there is still a common prefix in both traces:

1. The IF stages contain the same (potentially special, empty) instructions (PC) in the trace (tr) and in the other trace (line 4)—trace $3 - tr$ is the other trace, since $3 - tr = 2$ if $tr = 1$ and $3 - tr = 1$ if $tr = 2$;

2. If, in the next cycle (note the prime symbol), a certain instruction remains in the same stage in a trace, then this instruction also remains in the same stage in the other trace (implication lines 5-6).

Note that we use independent indexes (k and kk) since in superscalar versions, the IF stages play the same role and are indistinguishable. A similar check is also performed for the FUs (as indicated by the shaded comment). The treatment is exactly the same, in particular with two independent indexes, to allow for instructions that merely execute on another FU (see Sec. 5.2.4)—thus, the various FUs also act as indistinguishable (EX) stages. We do not consider the other hardware components for the comparison of traces, since they do not exhibit variable latencies and thus cannot cause a divergence, and, besides, state variable $commonPre$ remains FALSE when the traces have diverged.

Once the traces are about to diverge, the occupation of the various pipeline stages has to be compared in order to determine which traces represent a local

worst case. This is performed through state variable $locWorst$ (cf. Table 5.1), which contains a pair of Booleans, initialized to TRUE—traces are local-worst-case by default, see Sec. 4.2.4—, indicating for each trace whether it is local-worst-case:

$$
\begin{array}{ll}
locWorst' = [tr \in \{1,2\} \mapsto & 1 \\
\quad \lor \neg commonPre \land locWorst[tr] & \text{Remains local-worst-case trace if it was so} \quad 2 \\
\quad \lor \land commonPre & \text{Traces still comparable} \quad 3 \\
\quad\quad \land \forall\, k \in 1 \,..\, superscal : \forall\, kk \in 1 \,..\, superscal : & 4 \\
\quad\quad\quad IFs[tr][k].PC = IFs[3-tr][kk].PC & 5 \\
\quad\quad\quad\quad \Longrightarrow\ IFs[tr][k]'.currLat \geq IFs[3-tr][kk]'.currLat & 6 \\
\quad\quad \land \forall\, k \in 1 \,..\, N_{FU} : \forall\, kk \in 1 \,..\, N_{FU} : & 7 \\
\quad\quad\quad FUs[tr][k].PC = FUs[3-tr][kk].PC & 8 \\
\quad\quad\quad\quad \Longrightarrow\ FUs[tr][k]'.currLat \geq FUs[3-tr][kk]'.currLat] & 9 \\
 & 10
\end{array}
$$

This TLA^+ formula distinguishes two cases. In the first case (first disjunct, line 2), the two traces have already diverged ($\neg commonPre$); $locWorst[tr]$ then simply preserves its value for both traces (cf. Sec. 5.2.4). The second case (second disjunct, lines 3-9) considers the situation where the two traces have not diverged, in particular the situation where the two traces are about to diverge, i.e., $commonPre$ is still TRUE (line 3) but will be reset in the next cycle. At this moment, it is still possible to compare the occupation of the pipeline stages. Trace tr loses its status as a local worst case if, in the next cycle, an instruction in the other trace ($3 - tr$) has completed the fetch in IF (lines 4-6) or its computation in a FU (lines 7-9). We use the $FUs[tr][k]'.currLat$ and $FUs[3-tr][kk]'.currLat$ terms to detect a divergence in the next cycle due to a FU (this is similar for IF). The value of the $currLat$ field (line 9) of that FU (or IF stage, line 6) will thus be reset in this trace, resulting in diverging values in both traces that invalidate the implication.[10] Thus, the conjunction is FALSE and trace tr (line 1) becomes a non-local-worst-case trace. When both traces have not diverged and are not about to diverge, the last two conjuncts (lines 4-9) are true, either since the implications are vacuously true or since the latencies evolve identically in both traces. Note that we do not consider the other hardware components for the comparison of latencies, since they do not exhibit variable latencies *and* they are not suitable locations (Sec. 5.2.4).

With these two additional state variables ($commonPre$ and $locWorst$), it is possible to check for TAs, using the following invariant:

$$
\begin{array}{ll}
NoTALoc \triangleq \text{LET } n \triangleq Len(Program) \text{ IN} & 1 \\
\quad \land ProgDone(n) & 2 \\
\quad \land \neg locWorst[1] & 3
\end{array}
$$

[10] The implications are similar to those in the above action for updating $commonPre$, with the current latency in the stage instead of an instruction, and with inequalities associated to the use of two universal quantifiers (lines 4 and 7), due to the fact that we reason from the case where tr is *not* a local-worst-case trace.

$$\implies locWorst[2] \wedge ComTime(2,\, n) \geq ComTime(1,\, n) \quad 4$$

The formula might appear surprising at first sight, as one might expect a formula where $locWorst[tr]$ (with $tr \in \{1, 2\}$) simply implies that the global execution time in tr is the largest. However, both traces might be local-worst-case (cf. Sec. 4.2.4), and we have encoded the definition as it was stated in the original paper [22]. Note that this may entail different classifications of TAs, notably when both traces become local-worst-case due to opposing latency variations occurring at the same instant.

There is no TA iff, when the program is fully executed (line 2), if trace 1 is a non-local-worst-case trace (line 3), then trace 2 is a local-worst-case trace that has the same or a larger global execution time (line 4). Here again, the model checker will explore all pairs of traces, including where traces 1 and 2 merely switch their roles.

5.4 . Summary: the Parameters of our Formal Model

A formal model is essential to evaluate the existing definitions of TAs. Moreover, most of the definitions of TAs found in the literature are dissociated from concrete hardware microarchitectures and, thus, cannot be integrated as they stand into automatic tools. We thus specified precise assumptions to make the existing definitions applicable to an *automatic* detection of TAs on our formal model of the representative OoO-microarchitecture template. Our parametric model allows the exploration of FU affinities and variations in *latencies*, e.g., representing cache hits/misses, through execution constraints on the input program. It also allows the study of model variations, through the input program and/or microarchitectural parameters (a in Table 5.1). We defined helper operators and state variables (b and c in Table 5.1) to project the various definitions with their assumptions to our formal model, which enabled us to derive predicates that represent TA-detection procedures according to these definitions.

Table 5.1: Summary of model parameters (a), and helper operators (b) and state variables (c) used in the formalization of procedures based on the definitions of TAs for the OoO-pipeline template (cf. Sec. 2.3).

a	$program$	Input program with execution constraints about FU affinities and possible latencies in IF and FU (Sec. 5.1.2)
	$superscal$	Maximum number of instructions fetched/committed per cycle (Sec. 5.1.2)
	N_{FU}	Number of functional units (Sec. 5.1.2)
	S_{RS}	Size of the RSs (Sec. 2.3.1)
	S_{ROB}	Size of the ROB (Sec. 2.3.1)
	$locFU$	Subset of FUs considered for component occupation (Sec. 5.2.3)
b	$ProgDone(n)$	First n instructions were committed in both traces (Boolean, Sec. 5.3.2)
	$ComTime(tr, n)$	Commit instant of the n-th instruction in trace tr (Boolean, Sec. 5.3.2)
	$StepHeight(tr, n)$	$ComTime(tr, 1)$ for $n = 1$; $ComTime(tr, n) - ComTime(tr, n-1)$ otherwise (Sec. 5.3.2)
	$FUuse(tr, f)$	Number of cycles where FU f is occupied in trace tr (Sec. 5.3.2)
c	$commonPre$	The two traces have not diverged at current instant (Boolean, Sec. 5.3.2)
	$locWorst[tr]$	tr is a local-worst-case trace at current instant (Boolean, Sec. 5.3.2)

6 – ASSESSMENT OF THE DEFINITIONS

In this chapter, we use model checking in order to assess the various definitions of TAs, under the related detection procedures that we proposed in Ch. 5. First, we explain our approach for assessing the formal definitions by model checking and we provide a series of examples that highlight various shortcomings of the definitions (Sec. 6.1). Then, we conclude about this assessment, in particular by emphasizing the essential notion of causality (Sec. 6.2).

6.1 . Assessment by Model Checking

In this section, we present the verification methodology (Sec. 6.1.1) that allows us to make a comparative assessment of the definitions (Sec. 6.1.2).

6.1.1 . Verification Methodology

We do not intend to set up a procedure for detecting TAs over a wide range of programs, i.e., from a specific (reliable) definition. We invoke TLC, the model checker of TLA^+ (cf. Sec. 3.3.2), for particular instances of the parametric formal model (i.e., with all parameters fixed), in order to explore timing variations of otherwise identical traces, while evaluating the various predicates, as potential culprits for TAs. **Verifying the absence/presence of TAs in this way helps us to find inconsistent scenarios, e.g., where some definitions identify a TA for an instruction sequence while others do not.**

The state space that is to be explored mainly depends on parameter $program$ (cf. Table 5.1), namely the length of the input program, the program dependencies, and the specified execution variability through possible latencies and FU affinities. The depth of the state space is approximately the program length and the breadth is fully determined by the set of initial states. Indeed, the next-state relation of our specification is a *function* describing the deterministic advancement in the pipeline up to the completion of executions; choices (for actual latencies and assignments to FUs) are made in the initial state. Denoting as $IFLat$ and $FULat$ the sets of possible IF and, respectively, FU latencies (imposed in fact as execution constraints), checking the absence of TAs for *all* the input programs of length N would require exploring all possible dependencies ($\Pi_{k=0}^{N} 2^k \approx 2^{N^2/2}$), as well as all variations of IF and FU latencies for each instruction in each trace ($|IFLat|^{2N}$ and $|FULat|^{2N}$) and all FU affinities (${N_{\mathsf{FU}}}^{2N}$). Those terms essentially multiply and quickly result in a very large state space.

However, we do not need to consider all programs. We aim at getting **several scenarios, i.e., pairs of execution traces, that expose the contradictions**

among the definitions and their limitations. Those scenarios may differ from the input program and/or architectural parameters, e.g., $superscal$ and N_{FU}. Such scenarios are derived as *counterexamples* for specific—violated—properties. These properties are *invariants* expressing, for instance, that some definitions *always* make consistent statements about TAs (for fixed values of the parameters, thus for a fixed microarchitecture and a given program). We systematically *analyze* the obtained counterexamples to confront them to the intuitive understanding and then set up an invariant expressing another inconsistent scenario, e.g., a contradiction between another pair of definitions.

Our probation methodology starts from the basic example (in Fig. 5.1), then progressively proceeds through more elaborated variations in order to obtain convincing counterexamples that are short, easy to understand, and still illustrate a relevant shortcoming of at least one of the definitions. We supply program portions made of a few instructions and we perform progressive variations on the dependencies and on the constraints restricting the possible FU latencies and the FU affinities. In most examples, we only vary the FU latencies (since this is enough to demonstrate the shortcomings). In some cases, we also allow variations on the IF latencies to highlight specific shortcomings. Consequently, we do not face state space explosion and the worst complexity in our execution scenarios exposed in the next section is illustrated by a counterexample (cf. Fig. 6.5) requiring TLC to execute for 6 seconds and explore about 1,000 states. TLC provides counterexamples by enumerating the sequence of states that represent the execution scenario. Its output can be parsed automatically in order to obtain a visual illustration, e.g., the basis of all examples presented hereafter.

6.1.2 . Shortcomings of the Definitions

In this section, we present the (counter)examples found by TLC that we have retained for illustrating the shortcomings of the definitions. These traces are essentially based on the common example of Sec. 2.3.3, from which we investigate the impact of some additional instructions, data dependencies and/or execution variations on the stated presence/absence of TAs.[1] Note that we do not seek to unduly justify why the specific obtained traces are relevant, since we consider that any execution pair is suitable for stating TAs.

Importance of Structural Aspects

This example shows that taking into account the structure of the *whole* pipeline and its *microarchitectural features* is important to identify TAs. Let us resume the example in Fig. 5.1b and modify the value of parameter $superscal$ (from

[1] For this assessment, we use the version of the sources in branch `master` at: https://bitbucket.org/benjaminbinder/ta-models/ (cf. Ch. 5). Note that the assumption on infinite buffers in this version is reasonable for this assessment, due to the actual small length of input sequences (wrt. reasonable buffer sizes). The TLA$^+$ configurations of all examples are in the repository [2].

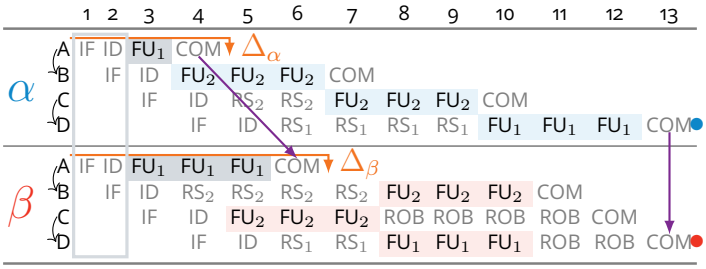


Figure 6.1: Importance of structural aspects.

$superscal = 2$ to $superscal = 1$) to allow at most one instruction per cycle in the in-order stages of our model. We only present the detailed table-based execution traces, which nevertheless contain all the information (and not the other representations, e.g., step functions). The execution traces, in both scenarios, are given in Fig. 6.1. Trace β takes longer compared to the dual-issue pipeline of Fig. 5.1b and all of the formal definitions correctly reflect the consequent intuitive absence of TAs. Though the definitions agree here, this first example confirms that: (i) the common scheduling diagrams (such as Fig. 2.4b), almost exclusively used in the literature, are *not* sufficient to study TAs; (ii) *executable* models of hardware platforms and program executions are beneficial for the concrete assessment of TAs.

Step Height and Execution Order

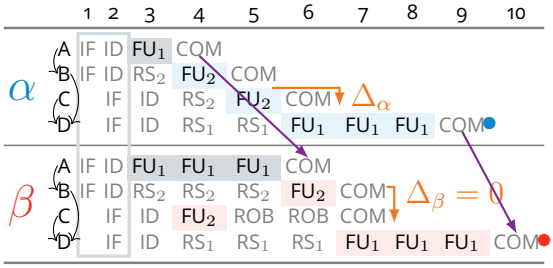


Figure 6.2: Step heights and execution order.

Both $TASteps$ and $TAInter$ require the completion of entire instructions, namely the observation of *commit events*. The slightly modified example of Fig. 6.2 shows that the step height of a specific instruction is likely to present TAs in unexpected situations. It is primarily obtained as a violation of the invariant $NoTAInter \implies NoTASteps$, i.e., the definition $TASteps$ states a TA though $TAInter$ does not. There is no clear counter-intuitive TA, since trace β with the 3-cycle latency in FU_1 leads to the WCET. Yet, (only) $TASteps$ indicates a TA. Indeed, the latency for instruction C in trace β is zero, because it is committed at the same time as B. While any inversion in cumulative execution times necessarily originates from variations in step heights at some points, this example shows that

the converse is not true. The step-height metric is too coarse-grained since values cannot be negative (due to in-order commit). **Consequently, it is not adequate to define and ultimately reason about TAs in terms of step heights.**

Intersections and Execution Order

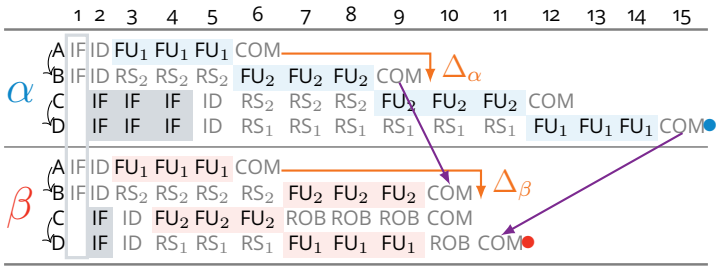


Figure 6.3: Intersections and execution order.

So far, we discarded only one definition among those based on commit events. The other definition, $TAInter$ seems, so far, to be more adequate. **However, this new example shows that surprising TAs arise from intersections too.** It is obtained with the violation of the invariant: $NoTALoc \implies NoTAInter$, where $TALoc$ is taken as an initial postulate. In the previous examples, no instruction memory accesses are performed (concretely plausible, for instance, with an instruction scratchpad). If instead we consider an instruction cache, a cache miss might increase the fetch delay. In Fig. 6.3, we present a pair of program executions with no variations in FUs, but a possible instruction cache miss for the last two instructions. Here again, there is no clear counter-intuitive TA, since the WCET is indeed given by the unfavorable scenario, i.e., with the instruction cache miss (trace α). This is confirmed by $TALoc$, which does not signal a TA (see the gray box and occupation). In the case of $TAInter$ (and $TASteps$), there is an evident inversion and thus a TA.

There is however a particularly surprising timing effect. The favorable case, i.e., the cache hit in trace β, entails a scheduling similar to that of trace β in the previous examples, namely instruction C starting its computation before instruction B, delaying the commit of B. Intuitively, the global execution of the unfavorable case α needs 4 additional cycles, whereas its cache miss shows a 2-cycle difference compared to the cache hit. This is an *amplification* effect that shows that both counter-intuitive and amplification TAs are closely related.

Deficiencies of Commit Events and Relevance of Locality

Showing unspecified behaviors wrt. the statement of counter-intuitive TAs is not the only shortcoming of $TAInter$. **The formulation based on intersections is also unable to detect all TAs.** The example from Fig. 6.4 shows that its high-level granularity based on commit events is insufficient; a finer control

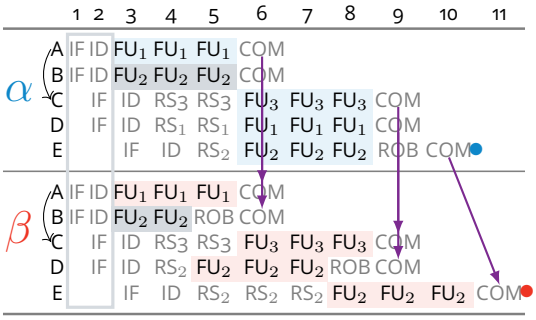


Figure 6.4: Commit events and relevance of locality.)

of pipelined executions (e.g., as in $TAComp$ or $TALoc$) is required. This example is based on the execution of a program with five instructions on a microarchitecture with $N_{\mathrm{FU}} = 3$. It is derived from the violation of the invariant: $(NoTAInter \vee NoTASteps) \implies NoTALoc$, assuming here that $TALoc$ is reliable for detecting TAs. Instruction B has a variable latency in FU_2 and instruction D can execute either on FU_1 or FU_2 (FU affinities). The two execution scenarios in the figure show choices of different FUs after a variable latency of instruction B (concretely plausible if instructions are preferably issued to FUs that are not busy).

The seemingly most favorable case, i.e., 2 cycles in FU_2 (trace β), eventually leads to the global worst case, which is intuitively a TA. Yet, only the last instruction differs in terms of commit events and hence the definitions based on commit events, $TAInter$ and $TASteps$, are unable to detect it. The definition $TALoc$ has the shorter trace α as a local-worst-case trace (and not trace β) due to instruction B, which does correspond to a TA. Similarly, $TAComp$ could detect a TA, though depending on a hardware partitioning. For instance, with the (sub)set $locFU = \{\mathrm{FU}_1, \mathrm{FU}_2, \mathrm{FU}_3\}$ (all highlighted cells), the TA is detected, due to the way that FU_2 is used.

Concern about Local Occupation for Applying Locality

We mentioned in Sec. 4.2.4 that a major concern of the original work from which we have defined $TALoc$ concerns the way of reasoning about the *local* occupation of the pipeline resources. However, the example in Fig. 6.5 shows that **the concern with local comparisons remains even with the clarifying hypotheses about locality constraints** added in Sec. 5.2.4. All instructions execute on FU_1 and, considering only traces α and β, trace β is the local-worst-case trace and there is no TA (whatever the definition).[2] Let us assume that the hardware model brings up (only) a third trace, α', in which the last two instructions experience instruction cache misses. Traces α' and β are concretely derived from the violation of the invariant: $(NoTAComp \vee NoTAInter) \implies NoTALoc$. The local-worst-

[2] Traces α and β are obtained from a property stating the absence of TAs from all definitions.

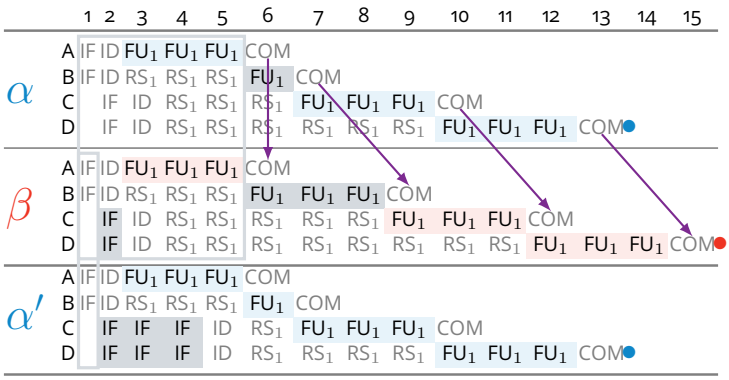


Figure 6.5: Comparing occupation of locations for locality.

case trace is now α', and $TALoc$ identifies a TA, due to the variation in instruction fetching. Note that with $TALoc$, we cannot properly compare the local occupation of FU_1 by instruction B for traces β and α', since these traces have diverged when B starts its computation on FU_1.

Actually, the variation in fetching (α vs. α') is independent of the one in the FUs, and it does not impact the scheduling on FUs. It is however clear that the verification based on commit events still states the absence of TAs, as well as when applying $TAComp$. **Comparing local resource use in this way is unreliable under more than one source of variations.**

Issue with Component Occupation

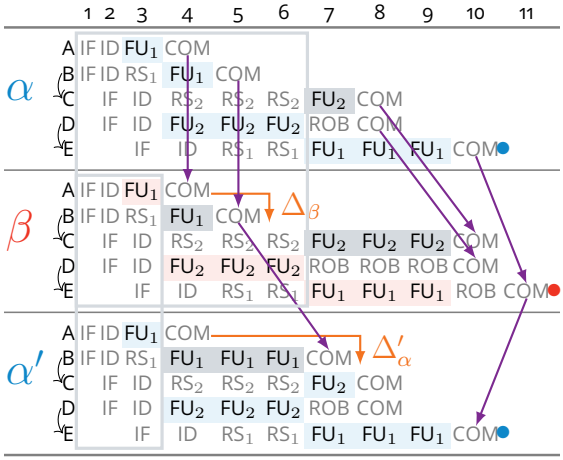


Figure 6.6: Component occupation.

We already showed that the arbitrary choice of relevant FUs for $TAComp$ has an impact on stating the absence of TAs (cf. Sec. 4.2.3). The example from Fig. 6.6 shows that **this definition based on the global use of components is not stable even with a preset hardware partitioning**. We consider two

independent variations in the computation latency of two instructions B and C. In the first place, we consider traces α and β, and we fix, for $TAComp$, the subset of FUs as $locFU = \{\mathrm{FU}_1\}$. Under $TAComp$, these traces do not present a TA, since the component occupation is 5 for both α and β—the other definitions of TAs yield the same answer.[2] However, the slightly modified trace α' emphasizes the previously introduced issue on comparing local resource use for $TALoc$, while $TAComp$ cannot address it: α' and β are merely derived from the violation of the invariant: $NoTAComp \wedge NoTALoc$.

When we consider traces α' and β, both $TALoc$ and $TAComp$ show the presence of a TA, since the component occupation in trace α' is 7. In any case, it is difficult to interpret the results of $TAComp$, since this definition gives absolutely no information whether a certain scenario is identified as a TA. Actually, wrt. α' and β, all definitions state a TA incriminating instruction B (because of its commit event or its FU latency). Yet, B cannot be the *cause* of a TA, since its execution variation (i.e., the latency in FU_1) is completely hidden by the execution of D. Specifically, instruction D is allowed to immediately start its computation in FU_2, so the computation of B in FU_1 does not alter the FU scheduling and C always starts its computation in cycle 7 (and, from that point, the single variation in FU_2 has no surprising effect). Intuitively, these independent variations do not generate a TA; however, no definition is able to separate the effects of the two variations (e.g., starting in cycle 7).

Issue about the End of the Instruction Sequence

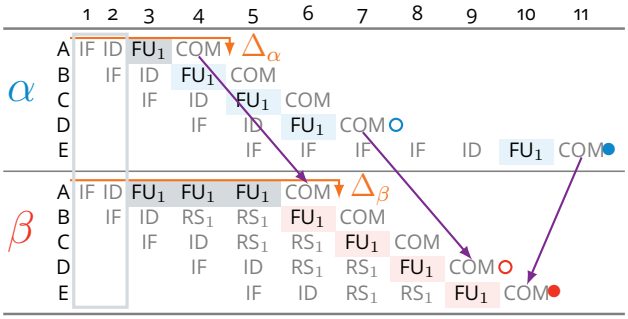


Figure 6.7: End of the instruction sequence.

Up to now, we focused on the main differences between the proposed criteria to define local variations, which allow us to define the favorable cases. The global comparisons are simpler to establish, since these are always based on the execution end of a certain *last* instruction, and, in our model, the end of an instruction is clearly defined by its commit event. However, all definitions rely on an instruction *sequence* within a program. The example in Fig. 6.7 shows that **the choice of the input sequence, thus of the last instruction for the global comparisons, is essential to properly assess TAs**.

Let us consider the simple example with a single FU and a single FU latency variation, in Fig. 6.7. This counterexample is derived from an invariant involving copies of the predicates that express the absence of TAs under the various definitions, in which, instead of the program length, we can specify through a parameter the last instruction to consider. If we do not consider the last instruction, the single variation drives the whole execution (with instruction A in FU_1) and all definitions confirm the absence of TAs. However, no definition can accommodate the fact that the last instruction(s) might add irrelevant extra cycles. If we consider the complete sequence (including instruction E), all definitions state a TA. It is interesting to remark that E is fully independent of the other instructions and no definition is able to capture the variation of this last instruction.

6.2 . Assessment Outcome

The systematic investigation of TAs showed that the definitions often lead to contradictory statements about TAs, even with themselves when we vary the last instruction of the considered input sequence. Thus, even under a precise evaluation framework, it is impossible to fix conditions under which a given definition behaves consistently and could serve as a reference in place of the intuitive understanding of TAs introduced in Sec. 1.3.1—in any case, in the intuitive and essential pairwise interpretation of TAs: **none of the existing definitions of TAs dominates the others**.[3]

Many examples show that the various definitions are not consistent when the traces exhibit several variations. The semi-formal definitions by Lundqvist and Stenström [20] and Wenzel et al. [21] explicitly assume that a single variable latency affects the trace comparison (cf. Sec. 4.2). However, no later formal definition restricts the way that traces may differ from each other, and the cases that we provided are not degenerated.

Moreover, we carefully analyzed the counterexamples to clarify the reasons why some definitions state a TA or not: **no definition is reliable when put to the test on an OoO pipeline**, i.e., none is always consistent with the intuitive understanding. We argue that this is due to the fact that no definition always adopts the suited pipeline-level granularity to define TAs (Sec. 6.2.1) and that all definitions lack the notion of causality to correctly reflect the intuitive understanding (Sec. 6.2.2).

6.2.1 . Unsuited Granularities for Detecting TAs

The definitions based on commit events only [80, 79, 24, 81], i.e., encoded by $NoTAInter$ and $NoTASteps$, which represent the instants where instructions leave the pipeline to define latencies and local variations, are unsafe: they are too coarse-

[3]The same holds when considering the definitions corresponding to $TAInter$ and $TAComp$ (by the same authors) complementary definitions of TA variants (i.e., with parallel inversions).

grained and omit relevant events in-between instruction commits, which may *hide* TAs and thus lead to inconsistent verdicts. Only the definition by Kirner et al. [23], encoded by $NoTAComp$, is not based on variations identified by instructions, but on a global favorable (i.e., lower) total use of certain resources that constitute a hardware partition. The detection of TAs is not conclusive, even after fixing a hardware partition, since one instruction may increase its use of a resource while another instruction decreases its use of the same resource, leading to the same total use in both traces. The definition by Reineke et al. [22], i.e., encoded by $NoTALoc$, is based on finer-grained comparisons of traces that allows capturing fine changes in the use of resources according to locality constraints. However, even when defining locality constraints with clear assumptions, the definition cannot always accommodate several variations. Besides, the granularity used for *global* comparisons is the same as in the other definitions, which may also lead to inconsistent verdicts (since we intend to analyze TAs independently of the WCET analysis).

6.2.2 . Towards the Notion of Causality

The examples show that the contradictions of the previously proposed formal definitions, corroborated with the intuitive understanding, stem from **a common deficiency: the notion of *causality***. These definitions are based on the presumed relation between local variations and global execution times. Yet, nothing ensures that a variation of a global execution time is *due to* the variation of an assessed local execution time. Such a causality link is however central in the intuitive perception of a TA. We observed that the definitions based on commit events can be easily manipulated by shifting the moment when a certain instruction ends, independently of surprising timing effects leading to TAs. The definitions based on components/locations may target local variations that would not be intuitively considered determining, as soon as an instruction sequence entails two (local) variation sources. The omission of causality is thus the main defect shared by all formal definitions. Note that, more generally, causality is required for finely capturing timing effects, even for cases with a single variation (cf. Fig. 6.3).

We exemplified shortcomings of the existing definitions of TAs on a representative microarchitecture and on simple, short instruction sequences. These definitions would be incomplete a fortiori on a more complex microarchitecture and larger programs. Larger programs might require defining start and stop instants to analyze a trace. Even in short examples (cf. Fig. 6.3), the notion of causality should prevent from arbitrarily slicing a trace. All the TAs identified above are interpreted through scheduling of instructions and depict the most commonly described class of TAs in the literature. However, TAs could also arise from speculation or cache effects, even in in-order pipelines (cf. Sec. 1.3.1). We believe that the notion of causality will be all the more relevant in these cases.

6.3 . Summary: the Lack of Causality

We assessed the various definitions of counter-intuitive TAs by model checking. The various exposed execution scenarios represent different situations (e.g., programs) reflecting plausible executions in an OoO pipeline and showing specific limitations of the definitions.

We showed that formal and executable models are essential to evaluate the existing definitions of TAs. We notably showed that common FU-scheduling diagrams are not sufficient to reason about TAs and that structural aspects must be taken into account. We also established that none of the existing definitions dominates the others, nor is it reliable to detect TAs on an OoO pipeline. We explained that a fine-grained, resource-level granularity is needed to capture fine variations, both in local and global comparisons of traces. The main deficiency shared by all definitions, in particular—but not only—in case of several sources of variations, is the lack of causality to relate global variations to their local causes.

Part III
Detection of Counter-Intuitive Timing Anomalies

None of the existing definitions of counter-intuitive TAs is able to correctly capture TAs on an OoO pipeline. These definitions share several issues. First, nearly all are based on hardware models that remain theoretical concepts, without a clear relation to the concrete hardware. As a consequence, they are often not implemented as TA-detection procedures or in any other practical setting. Second, they lack a way to correlate the local timing variations and their impact on global execution time. More precisely, we showed that the contradictions of the previously proposed formal definitions, corroborated with the intuitive understanding, stem from a common deficiency: the notion of *causality*. These definitions are based on the presumed relation between local variations and global execution times. Yet, nothing ensures that a variation of a global execution time is *due to* the variation of an assessed local execution time, even with a single local variation. Such a causality link is however central in the intuitive perception of TAs. Again, this shortcoming might explain the absence of tool support to reason about TAs.

Consequently, a precise formal definition of counter-intuitive TAs was still needed. We have integrated the crucial notion of causality into a *precise* and *practical* formal definition of counter-intuitive TAs, along with a detection procedure to prove the absence/presence of TAs. As stated in the previous part, a detection procedure necessarily relies on a specific hardware model. We propose a framework that lays the groundwork for the detection of counter-intuitive TAs—unambiguously applicable to a concrete microarchitecture and independent of any timing-analysis method:

1. **We propose a formalization of counter-intuitive TAs based on the notion of *causality*, which restricts the scope of a variation to the trace portion where the variation actually determines the timing behavior (Ch. 7) [3].**
2. **We instantiate this formalism on the *well-specified hardware model* representing an OoO pipeline (Ch. 7), with all the necessary information for the TA-detection procedure that we have implemented (Ch. 8) [3].**
3. **We evaluate the detection procedure of counter-intuitive TAs on the OoO-pipeline model wrt. false positives and faithfulness of the representation of scheduling effects established as *TA patterns* (Ch. 8) [3].**
4. **We apply our detection procedure on standard benchmarks, in order to pave the way for the concrete detection of TAs on real applications (Ch. 8).**

We also identify **a new problem, related to the *composition* of multiple variations**, since our accurate formalization of counter-intuitive TAs exposes more complicated scenarios on the traces under consideration. We consistently represent multiple variations individually, which allows for tackling this open problem.

CONTENTS

7 A Novel Formal Definition . . . 117
7.1 Reference Example . . . 117
7.1.1 Fine-Granularity Definition based on Locality . . . 118
7.1.2 Sketch of our Formal Definition . . . 119
7.2 Formal Definition of Counter-Intuitive TAs . . . 120
7.2.1 Execution Model for Timing Anomalies . . . 120
7.2.2 Event Time-Dependence Graph (ETDG) . . . 123
7.2.3 Relating Events between Traces . . . 126
7.2.4 Causality Graph (CG) . . . 127
7.2.5 Counter-Intuitive Timing Anomalies . . . 128
7.2.6 Application to the Reference Example (cf. Sec. 7.1) . . . 129
7.3 Correctness Arguments. . . . 132
7.3.1 Intuitive Understanding . . . 132
7.3.2 Prerequisites . . . 132
7.3.3 Formal Definition of Timing Anomalies . . . 135
7.4 Summary: a Groundwork for the Detection of TAs . . . 137
8 Detection Procedure . . . 139
8.1 Adaptation of the Formal Framework . . . 139
8.1.1 Formal Specification . . . 139
8.1.2 Property for the Absence of TAs . . . 141
8.2 Interpretation on Short Sequences . . . 145
8.2.1 Basic Variation Cases . . . 145
8.2.2 General Scenarios . . . 148
8.3 Detection of Timing Anomalies on Benchmarks . . . 151
8.3.1 Strategy and Heuristics . . . 151
8.3.2 Workflow . . . 154
8.3.3 Experimental Results . . . 155
8.4 Summary: a Tool Support for the Detection of TAs . . . 159

7 – A NOVEL FORMAL DEFINITION

In this chapter, we present our formalization of counter-intuitive TAs and its instantiation on the formal model of the OoO-pipeline template, introduced in Ch. 5. We use a slightly more complex version of the basic example of a counter-intuitive TA (cf. Sec. 2.3.3), in order to illustrate the key points of our proposed formalism (Sec. 7.1). Then, we expound our formalism, which results in a formal definition of counter-intuitive TAs (Sec. 7.2). We also provide some correctness arguments that allow gaining more confidence in our overall formalization approach (Sec. 7.3).

7.1 . Reference Example

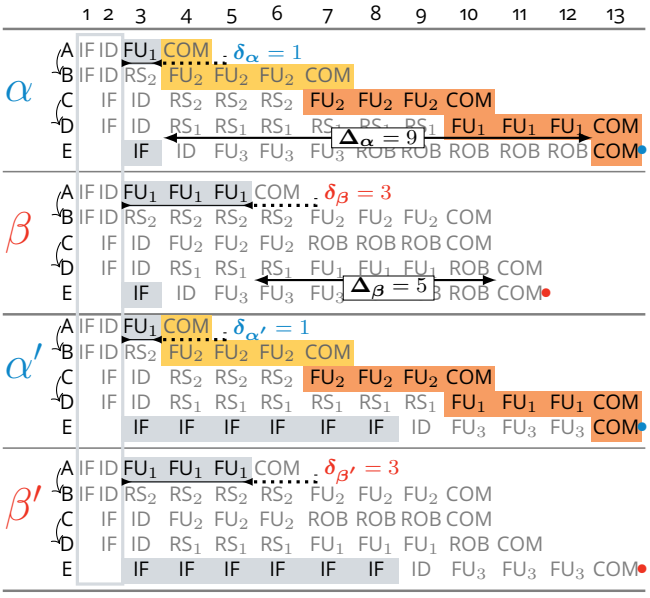


Figure 7.1: Execution of a program with data dependencies (↓) on our OoO-pipeline model (with $superscal = 2$ and $N_{\text{FU}} = 3$, cf. Sec. 2.3). Traces α/α' vs. β/β' exhibit a local variation wrt. FU_1 for A (▬). α/β vs. α'/β' exhibit a local variation wrt. IF for E (▬). β' is the local-worst-case trace according to the definition by Reineke et al. [22], which indicates the absence of TAs. α/α' vs. β/β', however, constitute a well-established example of a TA [21, 22, 23, 24, 34]. Symbols δ, Δ, and the colored cells are relevant for our definition (cf. Sec. 7.2).

We consider a sequence of instructions A to E with data dependencies, executed on our OoO-pipeline template (Fig. 2.3). The program is thus very close

to that in Fig. 2.4c, with additional, independent instruction E that uses a third functional unit ($N_{\mathrm{FU}} = 3$). Fig. 7.1 presents four traces, α, β, α', and β', corresponding to different pipelined executions of this sequence, from distinct initial hardware states yet the same input data. These traces may exhibit a local variation wrt. the use of FU_1 by instruction A. This particular instruction executes 1 cycle on FU_1 in α/α' (e.g., data-cache hit) and 3 cycles in β/β' (e.g., data-cache miss). These traces may also exhibit a local variation wrt. the use of the IF stage by instruction E, which requires 1 cycle in α and β (e.g., instruction-cache hit) and 3 cycles in α' and β' (e.g., instruction-cache miss).

The definition of counter-intuitive TAs by Reineke et al. [22] is the most appropriate wrt. the granularity of variations (cf. Sec. 6.2.1), since the underlying latencies are based on the **local occupation of a pipeline resource by an instruction**. In this section, we review the issues raised by this definition, when applied to this precise example (Sec. 7.1.1), before sketching from the example the functioning of the definition that we propose to address these issues (Sec. 7.1.2).

7.1.1 . Fine-Granularity Definition based on Locality

We thus focus on the definition based on locality (cf. Sec. 5.2.4), as a starting point for our approach and as a comparison basis. Hereafter, we recap from the example in Fig. 7.1 why, nevertheless, the absolute-WCET interpretation of this definition (cf. Sec. 4.1) and the presence of several sources of variations when comparing traces, without the notion of causality, are unsuitable for precisely detecting TAs.

Issue due to Pruning in the Absolute-WCET Interpretation

Since this definition is absolute-WCET, the same issue arises as in the example with a tri-valued variation (Sec. 5.2.4), due to multiple sources of variations (IF and FU_1). The four traces in Fig. 7.1 share the same prefix comprised of the first two cycles and then diverge due to two distinct, simultaneous variations in cycle 3, i.e., the use of FU_1 by A and of IF by E. Both variations identify the variations in β' as local worst cases (wrt. any other trace), thus this trace is a local-worst-case trace (cf. Sec. 5.2.4). Since this local-worst-case trace results in the global WCET, this definition does not signal a TA, as in the example from Fig. 5.2. However, a WCET analyzer might determine that trace β' is infeasible (e.g., the case with two cache misses is excluded). More precisely, the abstract state that opens up β' in cycle 3 should be pruned from the state space. A direct consequence of this pruning is that, in order to remain sound when applied, this definition should also address the remaining traces. Yet, it does not, since it follows only the local-worst-case traces (cf. Sec. 5.2.4).

The pruning under the local-worst-case trace leaves us with traces α, β, and α'. Let us inspect them closer. We note that instruction E does not impact the scheduling of the other (preceding) instructions because E has no data dependency and is the only instruction to use FU_3. Moreover, E does not impact the global execution time, which is, in fact, determined by the execution of instructions A

to D (i.e., the same in traces α and α'). We also note that the variation in the use of FU_1 by A, in traces α/α' as opposed to β, affects the scheduling of instructions B to D. This variation is favorable (i.e., A has a shorter latency) in α/α', and leads to a global slowdown wrt. β, notably for the commit of D and E. This case is the traditional TA pattern introduced in Sec. 2.3.3.

Issue due to Local Comparisons and the Lack of Causality

The definition by Reineke et al. [22] is unable to identify a local-worst-case trace among the remaining traces α, α', and β. Both variations are favorable for trace α, thus this trace cannot be a local-worst-case trace. Traces β and α' mutually prevent each other from being identified as a local-worst-case trace, since each trace has a variation that constitutes a local worst case that appears precisely when the traces diverge. However, the local worst case related to the use of FU_1 by A should serve as a basis to actually identify trace β as the local-worst-case trace, since it causes the particular scheduling of instructions B to D. This shows once again that this definition is limited in comparing traces with several sources of variations (cf. Sec. 6.2.2), thus for consistently reasoning about TAs.

We argue that a definition of TAs should be able to identify individual variations and to check whether these variations actually *determine* global slowdowns. Moreover, it should be able to identify chains of events from any favorable variation, defining trace portions of interest (▭/▭), later called *causal regions*. As such, a TA would be stated wrt. a trace if a slowdown is observed (▭) in the causal region. A formal definition that is able to identify variations and causal regions, to work under less restrictive assumptions (i.e., the existence of a static analysis to compute abstract hardware states) and to systematically discriminate between traces is introduced in Sec. 7.2.

7.1.2 . Sketch of our Formal Definition

Next, we provide an intuition on the way that our proposed formal definition (cf. Sec. 7.2) works on traces α and β (in Fig. 7.1) and addresses the issues highlighted above. Our definition is based on the pairwise interpretation of counter-intuitive TAs (cf. Sec. 4.1). We start from latencies ($\delta_\alpha = 1$ and $\delta_\beta = 3$) and then variations (▭), where we compute the *causal* region (▭/▭) in the trace where the variation is favorable, i.e., α ($\delta_\alpha < \delta_\beta$). This causal region covers events that are delayed directly or indirectly by the variation and thus could not occur earlier. The use of FU_2 by B and C, the use of FU_1 by D, and the commit (COM) of D form a *chain of events*, whose instants are *determined* by the variation of A on FU_1. It is only on this condition that we can say that the favorable variation in trace α *causes* a further slowdown. We then compute the relative time distance (↔) between the variation and each event of this causal region, for instance the commit of D ($\Delta_\alpha = 9$). We also compute the relative time distance between the variation and the corresponding events in the other trace ($\Delta_\beta = 5$ for the commit of D). If the relative time is greater in the favorable trace (i.e., in short,

the trace with the single favorable variation), a TA is triggered at this event (■). As indicated in Fig. 7.1, a TA is identified for the commit of D (■ in cycle 13 of α), which is in the causal region of the variation and has a greater relative time compared to β ($\Delta_\alpha > \Delta_\beta$).

In the following sections, we will define the necessary concepts: timing dependencies, latencies, corresponding events, variations, causality, causal regions, and relative time, based on Event Time-Dependence and Causality Graphs.

7.2 . A Consistent Formal Definition of Counter-Intuitive TAs

In this section, we gradually develop a formal definition of counter-intuitive TAs. We rely on two input **traces** derived from a transition system, from which we define **events** at the granularity of pipeline resources. The events represent the acquisition and the release of resources at some instants. From these events, we define an *Event Time-Dependence Graph* (ETDG) for each trace, whose arcs capture timing dependencies expressing the fact that a source event imposes a minimal duration before which another destination event cannot occur. The interval between the instants of the acquisition and the release of a resource by the same instruction defines a **latency**. From the ETDGs of both traces, we define **(favorable) variations** in the use of resources. Then, we introduce a *Causality Graph* (CG), a sub-graph of the ETDG where only the arcs that *exactly* and unambiguously explain the instant of the destination event remain. We then introduce the **causal region** of a favorable variation, as a sub-graph of the CG, to define the scope in which the variation determines the timing of other events. Finally, we combine all these elements to precisely capture counter-intuitive TAs triggered by the variation.

We first define general concepts and then provide concrete instantiations for our case study, the OoO-pipeline template (cf. Sec. 2.3). We focus on this precise microarchitectural model, providing modeling details.

7.2.1 . Execution Model for Timing Anomalies

Closely following the intuitive scheme outlined in Sec. 7.1.2, we first define execution traces, i.e., how the hardware executes a given sequence of instructions. The targeted hardware is a pipeline that may perform computations out of order. We only assume that the computed results are committed in-order—which is the case for modern processors.

Definition 7.1: Execution Traces — The set $\mathcal{T}$ of *execution traces* of a hardware model, represented by a transition system (TS), consists of all finite sequences of instructions executed by the TS, from any possible initial state.

Instantiation 7.1: Hardware Model & Execution Traces

We base our work on the formal OoO-pipeline specification developed in Sec. 5.1, from the template introduced in Sec. 2.3. In the following, we use the extended version of the model, with finite buffers (i.e., the ROB and the RSs), whose sizes are specified through model parameters (cf. Table 5.1).[a] From this formal specification, we can get execution traces similar to those represented in Fig. 7.1.

[a] The sources used in this part of the thesis, where the registers used as operands are explicitly stated in the input $program$ to detect data dependencies and where the (finite) ROB is modeled through a fully-fledged state variable that explicitly contain the instructions, are available in branch `detection_procedure` at: https://bitbucket.org/benjaminbinder/ta-models/

Remark — The hardware model is centered on the pipeline stages. In particular, the memory system is modeled implicitly, through variable latencies. Possible interference, e.g., on the memory bus, is thus not modeled. Other microarchitectural features (e.g., branch prediction, speculation) could also be considered, but we leave them for future work. This microarchitecture is prone to TAs, but sufficiently simple to reason about the relevant events that may occur during execution. This is also true for the choices of initial states—possible extensions are evoked in the conclusion.

In order to reason about TAs, we need to extract information from these execution traces. For this, we define two functions to capture the runtime behavior at the hardware level:

Definition 7.2: Events — Function *Events*: $\mathcal{T} \rightarrow \mathcal{P}(\mathcal{E})$ (where $\mathcal{P}$ denotes the power set) provides a set of triples $(i, r, t) \in \mathcal{E}$, called timestamped events, where i is an instruction identifier, r, a resource identifier, and t, a timestamp.

The *Events* function captures, from an execution trace (Def. 7.1), any timestamp t when an executed instruction i triggers an event associated with a resource identifier r, which may refer to an in-order commit unit or the acquisition/the release of a relevant hardware resource by an instruction.

Definition 7.3: Instruction Dependencies — Function *IDeps*: $\mathcal{T} \rightarrow \mathcal{P}(\mathcal{D})$ provides a set of triples $(i, j, t) \in \mathcal{D}$, where i and j are instruction identifiers and t a timestamp.

The *IDeps* function captures, from an execution trace, the data or control *dependencies* that impact an event of instruction j at timestamp t, due to instruction i.

Instantiation 7.2: Events & Instruction Dependencies

- In our case, instruction identifiers i are capital letters, e.g., A or B, according to the order of executed instructions.
- Instructions progress through the OoO pipeline; the fetch unit (within

the IF stage), the decode unit (ID) and the FUs are considered to be resources that are acquired when the instruction enters the unit and released when the instruction exits. Note that the release of a resource by an instruction corresponds to the cycle immediately *following* the last cycle where this instruction uses the resource.[a] For instance, in Fig. 7.1, the release of FU_1 by A in trace α occurs in cycle 4. The acquisition/release of one of these resources is denoted by an arrow pointing upward/downward, followed by the name of the unit/stage u: respectively, $\uparrow u$ and $\downarrow u$, e.g., $r = \uparrow \mathrm{IF}$ or $r = \downarrow \mathrm{FU}_1$. Instructions complete in the COM stage, which is reflected by a resource $r = \mathrm{COM}$. Each instruction is assigned an entry in the ROB/RS buffers in ID. The attribution of an entry is indicated by $r = \mathrm{ROB}$ and $r = \mathrm{RS}$ respectively.

- In our case, an instruction dependency of j on i may impact j at a single timestamp t, i.e., the timestamp of the acquisition of a FU by i. The *Events* and *IDeps* functions emit events/dependencies according to the progress of instructions in the pipeline as defined by Inst. 7.1.

[a] Then, the release of the resource by this instruction may coincide with the acquisition of the resource by another instruction.

The fetch unit and the FUs are considered relevant resources in our model, since they may have an intrinsic impact on the timing of other events. The time that an instruction spends in these resources thus represents a latency that can be explained directly by the *initial hardware state*. Note that $\uparrow / \downarrow \mathrm{IF}$ *exclusively* correspond to the time required for fetching an instruction (from memory or a bus): additional stalling may occur in the IF stage *after* $\downarrow \mathrm{IF}$, e.g., when the instruction in ID stalls. Also, the ID stage is relevant, since its timing is not determined only by the use of the fetch unit. The COM stage is relevant since it represents the completion of instructions, therefore a reference point when comparing traces. Since COM events are simple end-markers (they take a single cycle and may never cause stalling), there is no need to distinguish acquisition/release. Similarly, ROB and RS are markers within the ID stage (the behavior/content of ROB and RS are otherwise irrelevant for our approach).

Remark — While we focus on our OoO model, we would like to make some remarks relevant for more general architectures:

- We expect that events coincide with register writes in most cases, e.g., when data of an instruction is written into a pipeline register.
- On more complex architectures, it might not be sufficient to capture events only for pipeline stages. Events related to caches, buses, memories, etc. might be required.
- On real processors, it is not sufficient to fix the instruction sequence to be certain that the exact same program was executed, e.g., due to changes

in the input data. In this case, registers and memories visible through the instruction set architecture (defined by the programmer's manual or application-binary interface) have to be identical, whereas the behavior of hidden registers and memories may diverge (cf. Sec. 4.1).

7.2.2 . Event Time-Dependence Graph (ETDG)

The Event Time-Dependence Graph (ETDG) captures a minimal duration imposed between two events in a trace τ. The nodes are the events in the trace (Def. 7.2) and the arcs connect two nodes for which the source node *may* have a direct timing impact on the destination node.

Definition 7.4: Timing Dependencies of Events — Function *TDeps*: $\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{H})$ provides a set of triples $(e_1, e_2, w) \in \mathcal{H}$, called timing dependencies, where $e_1, e_2 \in \mathcal{E}$ are two events and $w \in \mathbb{N}$, a duration.

The *TDeps* function captures, from an execution trace, a set of minimal delays w between pairs of events (e_1, e_2) that suffices to explain (by transitivity) why any event of the trace *cannot* occur earlier, due to the structure of the trace and its resource use. This set of delays is derived from *microarchitecture-dependent rules* using information on the events of the trace (Def. 7.2), as well as on data dependencies among instructions (Def. 7.3).

Definition 7.5: Event Time-Dependence Graph (ETDG) — The ETDG of trace τ is the graph $G = (\mathcal{N}, \mathcal{A})$, where $\mathcal{N} = \mathit{Events}(\tau)$ is the set of nodes, which are directly derived from the events occurring during the execution of the trace (Def. 7.2), and $\mathcal{A} = \mathit{TDeps}(\mathit{Events}(\tau), \mathit{IDeps}(\tau))$ is a set of weighted arcs that specify *timing dependencies* (Def. 7.4).

An arc is denoted as $e_1 \xrightarrow{w} e_2$, where e_1 is the source event node, e_2 the destination node, and w the weight, i.e., a minimal delay imposed between the events.

Instantiation 7.3: Timing Dependencies & ETDG

Our microarchitectural model imposes order and delay constraints between events, captured by the following rules (which may be applied in any order).

1. **Order of pipeline stages**: The pipeline structure imposes a progression order wrt. a given instruction, as well as a minimal duration. In terms of events, any instruction X of a trace has nodes of the form $(X, \uparrow\mathrm{IF}, t_1)$, $(X, \downarrow\mathrm{IF}, t_2)$, $(X, \uparrow\mathrm{ID}, t_3)$, $(X, \downarrow\mathrm{ID}, t_4)$, $(X, \uparrow\mathrm{FU}_i, t_5)$, $(X, \downarrow\mathrm{FU}_i, t_6)$, and (X, COM, t_7). Instructions are decoded in a single cycle, so:

$$(X, \downarrow\mathrm{IF}, t_2) \xrightarrow{0} (X, \uparrow\mathrm{ID}, t_3) \in \mathcal{A}$$
$$(X, \uparrow\mathrm{ID}, t_3) \xrightarrow{1} (X, \downarrow\mathrm{ID}, t_4) \in \mathcal{A}$$
$$(X, \downarrow\mathrm{ID}, t_4) \xrightarrow{0} (X, \uparrow\mathrm{FU}_i, t_5) \in \mathcal{A}$$

Similarly, since an instruction might be committed immediately (i.e., when the preceding instructions are committed and a commit unit is free):

$$(X, \downarrow \mathrm{FU}_i, t_6) \xrightarrow{0} (X, \mathrm{COM}, t_7) \in \mathcal{A}$$

2. **Resource use**: The duration between the acquisition of the IF stage or a FU (by an instruction) and the matching release (by the same instruction), i.e., the duration of a resource use by the instruction, is determined by the *initial hardware state*. Hence, considering the events evoked in Rule 1, the weights of the related arcs are exactly the timestamp difference between events:
$(X, \uparrow \mathrm{IF}, t_1) \xrightarrow{t_2 - t_1} (X, \downarrow \mathrm{IF}, t_2)$, $(X, \uparrow \mathrm{FU}_i, t_5) \xrightarrow{t_6 - t_5} (X, \downarrow \mathrm{FU}_i, t_6) \in \mathcal{A}$

3. **Order of instructions in the input sequence**: If we consider two successive instructions in the input sequence, the stages of the in-order front-end and the in-order back-end (cf. Inst. 7.1) cannot process the second instruction before the first one. A minimal duration between these stages is 0: instructions could be processed at the same time (depending on the $superscal$ parameter). Any pair of *successive* instructions X and Y in a trace has events of the form $(X, \uparrow \mathrm{IF}, t_1)$ and $(Y, \uparrow \mathrm{IF}, t_1')$, which have to respect the program order, and thus:

$$(X, \uparrow \mathrm{IF}, t_1) \xrightarrow{0} (Y, \uparrow \mathrm{IF}, t_1') \in \mathcal{A}$$

The same applies for the decode/commit events: for nodes $(X, \uparrow \mathrm{ID}, t_2)/(Y, \uparrow \mathrm{ID}, t_2')$ and $(X, \mathrm{COM}, t_3)/(Y, \mathrm{COM}, t_3')$, it follows that:

$$(X, \uparrow \mathrm{ID}, t_2) \xrightarrow{0} (Y, \uparrow \mathrm{ID}, t_2'), \quad (X, \mathrm{COM}, t_3) \xrightarrow{0} (Y, \mathrm{COM}, t_3') \in \mathcal{A}$$

4. **Instruction dependencies**: In the OoO engine, the instructions use the resources independently of their order in the input sequence. However, the execution obviously respects data dependencies, which entails an order between *dependent* instructions. Moreover, back-to-back computations in FUs are possible (i.e., without delays, cf. Sec. 5.1.2). Any pair of instructions X and X' of a trace has events of the form $(X, \downarrow \mathrm{FU}_i, t)$ and $(X', \uparrow \mathrm{FU}_j, t')$ (possibly with $i \neq j$) since instructions always require a computation in a FU. If $IDeps(\tau)$ indicates that X' depends on a result produced by X, then:

$$(X, \downarrow \mathrm{FU}_i, t) \xrightarrow{0} (X', \uparrow \mathrm{FU}_j, t') \in \mathcal{A}$$

5. **Resource contention**

   (a) **Execution in a FU**: Even without dependencies, an instruction cannot be issued to its FU if another instruction is already using it. In this case, the instruction is ready but not executing, it has

to wait in the associated RS (cf. Inst. 7.1). Here again, back-to-back computations are possible when the FU is free (cf. Sec. 5.1.2). Such competing instructions have events of the form $(X, \downarrow \mathrm{FU}_i, t')$, $(Y, \downarrow \mathrm{ID}, t_1)$ and $(Y, \uparrow \mathrm{FU}_i, t_2)$, with $t_1 < t' \leq t_2$. If two instructions X and Y exhibit such events, then:

$$(X, \downarrow \mathrm{FU}_i, t') \xrightarrow{0} (Y, \uparrow \mathrm{FU}_i, t_2) \in \mathcal{A}$$

(b) **Limited in-order parallelism**: Instructions may also suffer resource contention in the in-order front-end and the in-order back-end, when more than $superscal$ instructions try to access the resources at the same time (cf. Inst. 7.1). This occurs when two successive instructions are not part of the same fetch bundle or when the second instruction is completed but not committed yet and thus remains in the ROB. Any pair of successive instructions X and Y has events of the form $(X, \downarrow \mathrm{IF}, t_1)$ and $(Y, \uparrow \mathrm{IF}, t_1')$. If $t_1 = t_1'$, then X and Y are not part of the same fetch bundle, s.t. Y is delayed and fetched when the resource is released:

$$(X, \downarrow \mathrm{IF}, t_1) \xrightarrow{0} (Y, \uparrow \mathrm{IF}, t_1') \in \mathcal{A}$$

Likewise, successive instructions have events (X, COM, t'), $(Y, \downarrow \mathrm{FU}_i, t_1)$ and (Y, COM, t_2). If $t_1 \leq t' < t_2$,[a] then Y is in the ROB and must wait for the end of the ongoing commit:

$$(X, \mathrm{COM}, t') \xrightarrow{1} (Y, \mathrm{COM}, t_2) \in \mathcal{A}$$

(c) **Finite resources**: Stalling occurs in ID whenever the capacity of the finite ROB or appropriate RS is reached (cf. Sec. 2.3.2). Otherwise, instructions remain a single cycle in this stage. If the ROB is full, ID is stalled until instructions are removed from the ROB: the new assignment in the ROB occurs *after* the instruction enters ID, and a minimal duration between the commit(s) that immediately precede the new assignment (within ID) and the end of the stalling in ID is 1. If the RS is full, ID is stalled until the end of an instruction's execution in the FU, and thus until an entry is freed in the RS: the new assignment in the RS occurs *after* the instruction enters ID, and a minimal duration between this execution end and the end of the stalling in ID is 0. Moreover, in both cases, the next instruction is transitively stalled in IF: a minimal delay of 0 is imposed between the end of the stalling in ID and the acquisition of ID by the next instruction.

Any instruction X has events of the form $(X, \uparrow\ \mathrm{ID}, t_1)$, (X, ROB, t_2), (X, RS, t_3), $(X, \downarrow \mathrm{ID}, t_4)$ and $(X, \uparrow \mathrm{FU}_i, t_5)$, with $t_1 \leq t_2 \leq t_4$ and $t_1 \leq t_3 \leq t_4 \leq t_5$. If $t_1 + 1 < t_4$, X is stalled in ID. If $t_1 < t_2$, then stalling occurs due to the ROB and an

instruction X' exists with an event $(X', \mathrm{COM}, t_2 - 1)$, so that:

$$(X', \mathrm{COM}, t_2 - 1) \xrightarrow{1} (X, \downarrow \mathrm{ID}, t_4) \in \mathcal{A}$$

If $t_1 < t_3$, then stalling occurs due to the RS and X' exists with an event $(X', \downarrow \mathrm{FU}_i, t_3)$, so that:

$$(X', \downarrow \mathrm{FU}_i, t_3) \xrightarrow{0} (X, \downarrow \mathrm{ID}, t_4) \in \mathcal{A}$$

Let Y be the instruction that follows X. It has an event $(Y, \uparrow \mathrm{ID}, t')$. If $t_1 < t_2$ or $t_1 < t_3$, then Y is transitively stalled:

$$(X, \downarrow \mathrm{ID}, t_4) \xrightarrow{0} (Y, \uparrow \mathrm{ID}, t') \in \mathcal{A}$$

[a] In the ideal progression, the release of a FU by Y (at t_1) coincides with the commit of Y (at t_2).

Remark — A similar reasoning would apply with additional events, resulting for instance from an explicit modeling of the memory system. Note that the number of events with the same timestamp is unlimited and that $\uparrow / \downarrow$ pairs can be nested. Thus, the rules reported above constitute a sound basis for more complex models.

The ETDG captures the use of resources by the instructions. This allows us to formally define latencies:

Definition 7.6: Latency — Given an acquisition event $(i, \uparrow u, t_\uparrow)$ and a matching release event $(i, \downarrow u, t_\downarrow)$, the *latency* δ of i wrt. that resource is $\delta = t_\downarrow - t_\uparrow$.

In our case, an arc always exists between these events (cf. Rule 2 of Inst. 7.3).

Our approach for modeling timing dependencies in a pipelined processor is similar to that of Li et al. [35] and Bai et al. [97] (cf. 4.4.2). However, the events of their data structures are not timestamped but are statically annotated with intervals representing *possible* latencies. Our ETDG serves as a basis to capture the causality emerging from *dynamic* effects, which may trigger TAs. Consequently, an ETDG refers to one *specific* trace and it contains runtime information for this trace, such as the actual instant of each event and the actual order imposed by contention (the EG by Li et al. [35], for instance, uses *undirected* arcs in this case).

7.2.3 . Relating Events between Traces

Henceforth, we consider two traces α and β that execute precisely the same instruction sequence and for which we want to decide whether a TA exists or not. As such, we need to be able to reason about events that occur in both of those traces and relate events from one trace to events in the other trace:

Definition 7.7: Corresponding Event — Function *CospEvent*: *Events*$(\alpha) \rightarrow$ *Events*(β) maps an event of trace α to its *corresponding event* of trace β.

Instantiation 7.4: Corresponding Event

For the microarchitectural model from Inst. 7.1, such a mapping is straightforward. The acquisition/release or occupation related to the IF, ID, and COM stages of a given instruction identifier are simply mapped to the same events of the other trace of the same instruction identifier, i.e., an event $(i, r, t_\alpha) \in \mathit{Events}(\alpha)$ is mapped to $(i, r, t_\beta) \in \mathit{Events}(\beta)$. However, instructions may execute on different FUs in the two traces. For events related to FUs, we thus simply map to that other FU, i.e., $(i, \uparrow \mathrm{FU}_\alpha, t_\alpha) \in \mathit{Events}(\alpha)$ is mapped to $(i, \uparrow \mathrm{FU}_\beta, t_\beta) \in \mathit{Events}(\beta)$ (similarly for the release of FUs).

Remark — Note that on our hardware model, such a mapping always exists, i.e., for every event in one trace, a corresponding event exists in the other trace. This might not be the case for all models, e.g., when the bus or memory is not accessed due to a cache hit. In that case, the *CospEvent* function needs to be adapted accordingly.

Since the corresponding events between the two traces are used to compare latencies (Def. 7.6), we can define variations that represent a favorable local case:

Definition 7.8: Variation — Let δ_α be the latency of a given instruction wrt. a given resource (Def. 7.6) in trace α and δ_β be the latency obtained from the corresponding events (Def. 7.7). We observe a *variation* if $\delta_\alpha \neq \delta_\beta$, more precisely a *favorable variation* for α (β) when $\delta_\alpha < \delta_\beta$ ($\delta_\beta < \delta_\alpha$).

Similarly we can detect whether an instruction has switched from one functional unit to another:

Definition 7.9: Resource Switch — A *resource switch* occurs when the corresponding events (Def. 7.7) of an instruction's resource use in trace α refers to a different resource in trace β, i.e., for $e = (i, r_\alpha, t_\alpha) \in \mathit{Events}(\alpha)$ and $\mathit{CospEvent}(e) = (i, r_\beta, t_\beta)$ we have $r_\alpha \neq r_\beta$.

In our OoO model, variations arise from the resources IF and FU_i, and resource switches only from FUs.

7.2.4 . Causality Graph (CG)

Now that we can build the ETDGs of traces α and β, which capture the order as well as timing dependencies among events of the traces, we further refine the graphs in order to capture the causality.

Definition 7.10: Causality — Function $\mathit{Causality} : \mathcal{G} \times \mathcal{G} \to \mathcal{P}(\mathcal{H})$, where $\mathcal{G}$ is the set of all ETDGs, provides, for a pair of ETDGs, the subset of timing dependencies in the first trace of the pair where the source node has a *direct impact* on the destination node in terms of timing.

Generally, not all arcs of an ETDG correspond to this notion of causality, and, consequently, some arcs are removed from the graph, resulting in the causality graph:

Definition 7.11: Causality Graph (CG) — Given ETDGs $G_\alpha = (\mathcal{N}, \mathcal{A})$ and G_β of both considered traces (Def. 7.5), the CG is the sub-graph $C = (\mathcal{N}, \textit{Causality}(G_\alpha, G_\beta) \subseteq \mathcal{A})$ of G_α, where only the arcs that reflect *causality* (cf. Def. 7.10) of events, called causal arcs, are retained.

Causal arcs $e_1 \xrightarrow{w} e_2 \in \textit{Causality}(G_\alpha, G_\beta)$ represent situations where event e_1 in trace α has actually a direct timing impact on event e_2 of the trace, i.e., the first event *determines* the timestamp of the other event. Node $e_1 \in \mathcal{N}$ is causal to node $e_2 \in \mathcal{N}$ iff there exists an arc $e_1 \xrightarrow{w} e_2 \in \mathcal{A}$ that is causal.

Instantiation 7.5: Causality & CG

For our OoO model (Inst. 7.1), we distinguish three rules, identifying, from ETDGs G_α and G_β, the cases where an arc $a = e_1 \xrightarrow{w} e_2 \in \mathcal{A}$ between two events $e_1 = (i_1, r_1, t_1), e_2 = (i_2, r_2, t_2) \in \mathcal{N}$ has to be removed, i.e., $a \notin \textit{Causality}(G_\alpha, G_\beta)$:

1. *Timing gap*: An arc has to be removed when $t_1 + w < t_2$. In this case, another event should exist that delays e_2 more than the duration w due to e_1, so that e_2's timestamp is not determined by e_1 (at least not via that arc of the ETDG).

2. *Variation*: An arc needs to be removed if it corresponds to a variation (thus e_1 represents a resource acquisition and e_2 the matching release) (Def. 7.8). Any event e_0 that occurred before e_1 and that is causal wrt. e_1 is no longer causal to any event e_3 that occurs after e_2 (even if e_2 is causal wrt. e_3), since the timestamp of e_3 is not only determined by e_1 but also by the variation that lies between them.

3. *Resource switch*: The same occurs when an instruction switches from one FU to another (Def. 7.9). The assignment to a FU results from the initial state (cf. Inst. 7.1), and consequently, this choice also determines the timestamp of later events e_3 according to the scheduling on FUs.

In particular, if an arc is causal, then the minimal duration specified by its weight is the actual timestamp difference between both events (Rule 1). Note that the CG of a specific trace may vary, depending on the other trace under consideration (due to Rules 2 and 3).

Definition 7.12: Causal Region — Given a causality graph $C = (\mathcal{N}, \textit{Causality}(G_\alpha, G_\beta))$ (Def. 7.11) and an event $e \in \mathcal{N}$, we define the *causal region* of that event, denoted by $C(e)$, as the sub-graph obtained from the nodes that are reachable from e. The set of nodes of $C(e)$ is denoted as $\mathcal{N}_{C(e)} \subseteq \mathcal{N}$.

7.2.5 . Counter-Intuitive Timing Anomalies

Based on the variations and their causal region, we can now reason about TAs. We formally define them in accordance with the intuitive definition. Nevertheless:

1. The definition is based on a precisely defined variation (Def. 7.8) in how an

instruction uses a *resource*.

2. The causal region (Def. 7.12) of this variation limits the *scope* of the TA verdict to this region (i.e., not necessarily the whole trace).

3. Contrary to previous definitions, we do not rely on the (absolute) global execution time. Instead, we compare the *relative* time distance of events by using the operator Δ, which computes the time distance $\Delta(e_1, e_2) = t_2 - t_1$ of two events $e_1 = (i_1, r_1, t_1)$ and $e_2 = (i_2, r_2, t_2)$.

Definition 7.13: Counter-Intuitive Timing Anomaly — For $\tau = \alpha$ or $\tau = \beta$, let $e_{\tau\uparrow} = (i, \uparrow r_\tau, t_{\tau\uparrow})$ be an acquisition event and $e_{\tau\downarrow} = (i, \downarrow r_\tau, t_{\tau\downarrow})$ be the matching release event, s.t. $e_{\beta\uparrow} = \mathit{CospEvent}(e_{\alpha\uparrow})$ and $e_{\beta\downarrow} = \mathit{CospEvent}(e_{\alpha\downarrow})$. Event $e_{\alpha\downarrow}$, which represents the release of a variation in α, triggers a *counter-intuitive* TA at event e wrt. β, iff:

1. **Variation:** α exhibits a *favorable variation* (Def. 7.8) at $e_{\alpha\downarrow}$, i.e.:
$$(\delta_\alpha = t_{\alpha\downarrow} - t_{\alpha\uparrow}) < (t_{\beta\downarrow} - t_{\beta\uparrow} = \delta_\beta)$$

2. **Causality:** e is a node of the *causal region* $C(e_{\alpha\downarrow})$ of the variation (Def. 7.12):
$$e \in \mathcal{N}_{C(e_{\alpha\downarrow})}$$

3. **Slowdown:** α exhibits a *relative* slowdown, expressed as:
$$\Delta(e_{\alpha\downarrow}, e) > \Delta(e_{\beta\downarrow}, \mathit{CospEvent}(e))$$

We can obviously apply the definition with α and β exchanged in order to get TAs for favorable variations in β.

While this definition applies to all events e, we notably focus on COM events. Such events are relevant since the related instructions are fully executed and can no longer impact the execution of other instructions in the trace. However, considering terminal nodes other than COM events, i.e., nodes without any successors in the causal region, is particularly relevant for events representing a resource switch or another variation. This is necessary to reason about the *composition* of variations and about chains of TAs, as in certain examples presented in the next chapter. In any case, ROB and RS events explain the whole scheduling but not TAs directly.

7.2.6 . Application to the Reference Example (cf. Sec. 7.1)

Next, we present in further detail how the various definitions/instantiations work on the reference example (cf. Sec. 7.1), and how our definition addresses the raised issues. As a first step, we consider only traces α and β. Fig. 7.2 shows the ETDG and CG of this pair of traces—all arcs are in the ETDG while the dashed arcs are not in the CG, which contains only the solid arcs.[1] The successive steps in the application of our definition together constitute a TA-identification *procedure*.

[1]These graphs are generated automatically, from the framework described in Sec. 8.1.

1. The first step consists in extracting *events* from both considered traces of Fig. 7.1. The derived events (cf. Def. 7.2/Inst. 7.2) are the nodes in Fig. 7.2a and 7.2b.

2. From these events and from the time-dependence rules of Inst. 7.3, we build the ETDG (Def. 7.5) of each trace. In Fig. 7.2a and 7.2b, the nodes that have the same timestamp are vertically aligned. The arcs derived from Rule 1 (order of stages) and Rule 2 (resource use) are represented with bold black arrows ($\rightarrow$), those from Rule 3 (order of instructions) with simple black arrows ($\rightarrow$), those from Rule 4 (instruction dependencies) with red arrows ($\rightarrow$), those from Rule 5a (contention in a FU) with blue arrows ($\rightarrow$) and, finally, those from Rule 5b (limited parallelism) with green arrows ($\rightarrow$).

3. Both ETDGs exhibit a single *variation* (Def. 7.8), namely from the latencies (Def. 7.6) related to the use of FU_1 by instruction A in both traces, denoted as δ_α and δ_β. The variation is highlighted similarly in Fig. 7.1 and 7.2 (▬). The variation is favorable for α ($\delta_\alpha = 1 < 3 = \delta_\beta$).

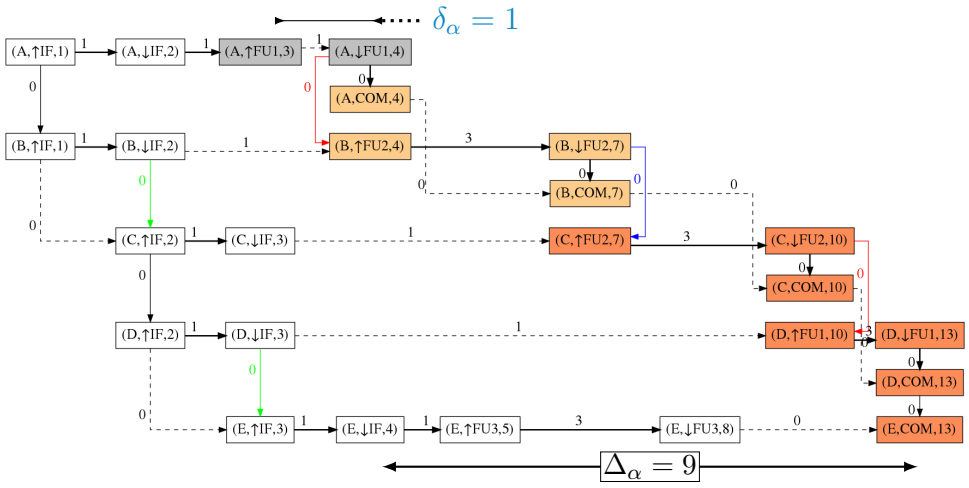
4. Both CGs (Def. 7.11) are derived from the ETDGs by removing the dashed arrows in Fig. 7.2, according to Rules 1 and 2 of Inst. 7.5. Rule 3 does not apply due to the absence of resource switches.

5. We then compute the *causal region* $C(e_{\alpha\downarrow})$ (Def. 7.12) of the release event $e_{\alpha\downarrow} = (A, \downarrow \mathrm{FU}_1, 4)$ using the CG C of α, which contains the favorable variation. The nodes of this region are highlighted in Fig. 7.2a (▬/▬)—reflecting the same information as in Fig. 7.1.

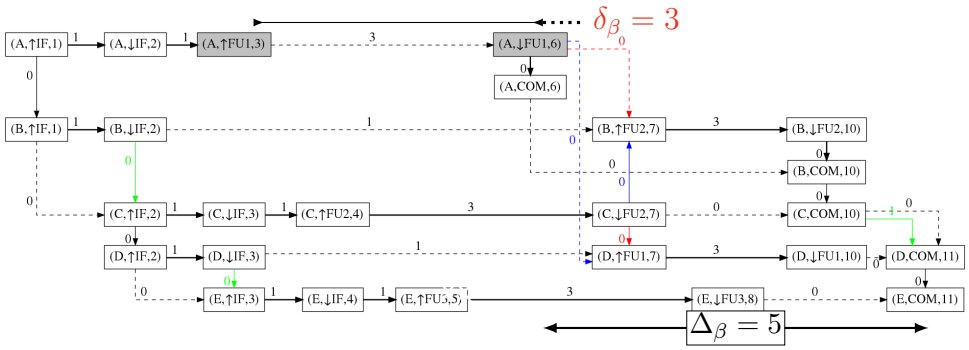
6. We finally compare the *relative time distance* from $e_{\alpha\downarrow}$ to each event $e \in \mathcal{N}_{C(e_{\alpha\downarrow})}$ of the causal region, with the relative time distance from the corresponding release event $e_{\beta\downarrow} = \mathit{CospEvent}(e_{\alpha\downarrow}) = (A, \downarrow \mathrm{FU}_1, 6)$ to each corresponding event $\mathit{CospEvent}(e)$ in trace β. The events at which a TA manifests (Def. 7.13) are highlighted with a more pronounced color in Fig. 7.1 and 7.2a (▬). Let us consider, in particular, the commit event $e_E = (E, \mathrm{COM}, 13) \in \mathcal{N}_{C(e_{\alpha\downarrow})}$.
   The relative time distance is $\Delta(e_{\alpha\downarrow}, e_E) = 9$, denoted by Δ_α in Fig. 7.1 and 7.2a. The corresponding relative time distance in the ETDG of β is $\Delta(e_{\beta\downarrow}, \mathit{CospEvent}(e_E)) = 5$, with $\mathit{CospEvent}(e_E) = (E, \mathrm{COM}, 11)$, denoted by Δ_β.[2] Trace α is longer ($\Delta_\alpha > \Delta_\beta$), thus a TA is triggered by $e_{\alpha\downarrow}$ at e_E.

Let us now bring α' up, so that we consider the three traces that remain after pruning β' from the reference example. The variation related to the use of IF by instruction E is favorable in α and β against α'. It can be observed from

[2] $\mathit{CospEvent}(e_E)$ may not be in the causal region of the variation in β.



(a) Trace α



(b) Trace β

Figure 7.2: ETDG/CG of α and β from the reference example (cf. Fig. 7.1). For each trace, all arcs are in the ETDG, while the arcs represented with dashed arrows are not in the CG. The nodes of the causal region of the favorable variation are highlighted (■/■). Among these nodes, those at which a TA manifests are highlighted in a more pronounced manner (■).[3]

Fig. 7.1 that in α and β, the causal region of the release event of this variation is limited to the use of FU_3 by E. There is no slowdown in these causal regions wrt. α', and thus no TA is signaled. This correctly reflects the fact that for the considered traces, this variation has no scheduling impact on the other instructions (cf. Sec. 7.1). Besides, the variation in FU_1 is favorable for α' wrt. β. The causal region of the release event in α' is exactly the same as in α, thus entailing the same TAs wrt. β. **This consistently captures the TA pattern shared by both traces α and α'.**

[3]For readability, we do not represent ROB and RS nodes in the figures (cf. Sec. 7.2.5). Moreover, for simplification, we do not represent ID nodes in this figure (no stalling occurs here in ID).

Finally, our definition does capture the TA pattern even if we consider β' instead of β. In α/α', exactly the same TAs as wrt. β are triggered wrt. β', for two reasons stemming from the fact that our definition is not based on the WCET: we do not exclusively focus on the end of the traces, and moreover we compare relative times.

7.3 . Correctness Arguments

In this section, we intend to show that the identification of TAs from our definition of counter-intuitive TAs on the OoO model, based on the various instantiations from Sec. 7.2, is accurate wrt. the intuitive understanding of TAs, which is reminded below in terms of events.

7.3.1 . Intuitive Understanding

TAs can be intuitively defined as follows, from informal definitions of fundamental notions: execution trace, latency, (favorable) variation, local/global effect, speedup/slowdown. A *trace* is a detailed execution scenario showing the hardware elements (i.e., resources) used when executing a program. TAs may occur when *comparing* two traces of the exact same instruction sequence executed from the same input data, nevertheless from different initial hardware states. One can observe events in such traces, e.g., the start of a computation performed by an instruction. Certain events can be compared in both traces, since they represent how an instruction uses certain resources. A *latency* is the duration between two particular events in a trace, for instance identifying a resource use by an instruction. The two traces considered for TAs may present a variable latency related to comparable events in both traces (e.g., a cache hit or a cache miss), depending on the initial hardware state. This is often called a *local variation*. The trace in which the latency of the local variation is smaller (greater) thus constitutes a *local speedup* (slowdown). The local variation is *favorable* for the trace with the local speedup. The execution time up to a certain later event (e.g., the end of the trace) may be qualified as *global*. We can also compare this global execution time in both traces and derive a *global slowdown* in one trace. A TA is thus stated when a local speedup and a global slowdown occur in the same trace, wrt. the other trace, reflecting that the processor performs more work *sequentially* after the favorable variation. We consider slowdowns emerging from instructions being scheduled for execution in our OoO-pipeline model.

7.3.2 . Prerequisites

We first investigate the correctness of the ETDG, which establishes a link between the actual execution of the input traces on the OoO model (Inst. 7.1) and our formalization.

Lemma 1: ETDG Accuracy — The ETDG (Inst. 7.3) is **accurate**, i.e., 1. its nodes exactly represent the relevant observable events of an execution trace for the study of TAs; 2. for all pairs of events for which an order is imposed by the OoO model during the execution of a trace, arcs exist in the graph; 3. the arc weights represent actual delays imposed by the OoO model between the respective events.

PROOF . *1.* COM *events are indispensable in order to identify executions of the* same *program and to delimit the contribution of a given instruction within an execution of our Transition System (*TS*). Besides, the acquisition/release of resources (*IF*/*FU*) that provide information about the* initial hardware state *must be represented, since the intuitive understanding of* TA*s relies on latencies ensuing from the initial state. The observation of the* COM *stage (cf. Sec. 7.2.1) is subject to insignificant simplifications, since commit always takes* 1 *cycle. The duration spent in the* RS*s and the* ROB *is an* emerging *property of the pipeline scheduling, not a latency that is directly related to the initial state.*

2. For brevity, we do not provide a full proof showing that the ETDG *reflects any possible evolution of the state variables that represent relevant resources, specified by the transition relation of our* TS*. However, the microarchitectural model specified from Inst. 7.1, on the basis of the template described in Sec. 2.3, may impose an ordering among events only in four situations (Rules 1/2 and 3-5 of 7.3). Note that it is sufficient to consider the* direct *timing dependencies. Other (later) events may be impacted by transitivity.*

3. A similar analysis yields the correctness of the weights of arcs in the ETDG*, which represent delay constraints on the observed time distance between events.* ■

The intuitive definition of TAs from Sec. 7.3.1 furthermore requires that traces are *comparable*.

Lemma 2: Trace Comparison — Inst. 7.4 as well as Def. 7.8 and 7.9, which build on it, allow a **consistent comparison** of the two traces at hand.

PROOF . *The relevant notions (instructions, resource use) are captured by events (Def. 7.2) and latencies (Def. 7.6). The correctness follows immediately from the design of the microarchitectural model. Notably, variations and resource switches are well defined due to the fact that a unique resource use can be identified (via the corresponding events) in the respective other trace.* ■

The objective of the CG is to capture *causality*, i.e., the relationship of two events within a trace s.t. the first event *explains* why the second event occurred at a specific instant.

Lemma 3: CG Accuracy — The CG (Def. 7.11) is **accurate**, i.e., 1. all arcs of the graph link causal events; 2. all the events that verify the causality relationship are connected through an arc.

PROOF . *1. We must ensure that all remaining arcs in the* CG *correspond to the causality relationship. To do so, we must verify that Inst. 7.5* exactly *characterizes this relationship. All arcs are also in the* ETDG*, so they represent a delay constraint. This constraint is clearly necessary to establish the causality but it is not sufficient. We must verify that the source event* e_1 determines *the timestamp of the destination event* e_2*. Indeed, a timing gap may exist due to the timing dependency of* e_2 *on another event* e_0*. Inst. 7.5 (Rule 1) ensures that such an arc between* e_1 *and* e_2 *is not present in the* CG*. Note that in this case, an arc must exist between* e_0 *and* e_2*, due to the second item of the instantiation.*
Besides, we must verify that the causality relationship takes into account the comparison of both traces at hand, since we are interested in explaining *whether a given event caused a divergence between the two traces. Variations and resource switches are the only source of divergence between the execution of traces (cf. Inst. 7.1), and they are correctly captured (cf. Lemma 2). All other effects (e.g., the order of computations on* FU*s) are emerging from that in the hardware model. Recall that variations are based on latencies and that the assignments to* FU*s are independent and only depend on the initial hardware state (cf. Inst. 7.1). Consequently, the instants of the events that are time-dependent on the release* e_2 *of a resource use that exhibits a variation or a switch are not explained by the matching acquisition* e_1 *alone. Rules 2 and 3 of Inst. 7.5 complete the removal of undesired arcs, without loss of relevant information (the* ETDG *suffices to characterize the resource use in question).*

2. Assume that a pair of events exists that verifies the causality relationship, but no arc in the CG *connects them. By definition, the* CG *is a sub-graph of the* ETDG *that shares the same event nodes. An ordering exists between the two events. Consequently, a path must exist between the two nodes in the* ETDG *(due to Lemma 1) and at least one arc along this path was removed during the* CG *construction. The removed arc either represents a timing gap or a variation/resource switch. In the former case, the initial hypothesis on causality is contradicted. In the latter case, the initial variation is no longer the only explanation for the instant of the destination event. Consequently, an arc must exist between the two considered events.* ■

Causal regions represent *chains* of events where each event is time-determined by its predecessor.

Lemma 4: Timing in Causal Regions — Given a causality graph C, for any pair of events (e_1, e_2), where $e_2 \in \mathcal{N}_{C(e_1)}$ (Def. 7.12), the relative time distance $\Delta(e_1, e_2)$ between the two events **corresponds exactly** to the sum of the arcs weights on any path between the two events of $C(e_1)$.

PROOF . *This follows from Lemma 3 and by induction from the fact that any arc of* $C(e_1)$ *satisfies Rule 1 of Inst. 7.5.* ■

Lemma 4 indicates that causal regions capture the desired notion that the source event (here e_1) determines the instant of the destination event (e_2), i.e., e_2 could not occur earlier due to e_1.

Lemma 5: Resource Use in Causal Regions — Considering a causal region (Def. 7.12), all the instructions involved in the events of this region make the **same use of resources** involved in these events in both traces, i.e., they use the same resources, with the same latencies.

PROOF . *The proof is similar to that of 4. It is immediate from an inductive reasoning on arcs with Rules 2 and 3.* ■

Lemma 5 indicates that the difference between both traces regarding the events of a causal region may only reside in *emerging* properties from the scheduling, namely the other trace makes the same use of resources, possibly in a different order or even in parallel.

7.3.3 . Formal Definition of Timing Anomalies

We now argue that Def. 7.13 corresponds to the intuitive definition of TAs exposed in Sec. 7.3.1, considering the notion of causality. We will proceed in two steps:

1. considering scenarios with a **single variation** and without resource switches;
2. considering **general scenarios** with possibly many variations and/or resource switches.

First, however, we need to discuss our choice to rely on *relative time* instead of absolute time. The main reason for this is due to the fact that absolute time cannot reliably serve as a reference once the two traces have diverged. This has been shown to lead to inconsistent verdicts for various definitions (cf. Ch. 6).

Single Variation Most existing definitions are limited to this kind of execution scenarios (often without explicitly stating so).

Lemma 6: Counter-Intuitive TAs — For the OoO model from Inst. 7.1, Def. 7.13 corresponds to the intuitive understanding of TAs, as stipulated in Sec. 7.3.1.

PROOF . *Given the vagueness, inherent to the* intuitive *understanding of* TA*s, it is impossible to provide a formal proof. We will thus develop a series of* arguments *highlighting different aspects of the definition and showing that its verdicts are coherent with this intuitive notion of* TA*s.*
We first investigate the three necessary conditions at the heart of our definition, by assuming the absence of each of them:
1. ***Variation:*** *Suppose that the two input traces do not exhibit any variation. Consequently, the two traces are identical and our definition does not signal a* TA*—conforming to the intuitive notion of* TA*s.*
2. ***Causality:*** *Now suppose that a (single) favorable variation is present in one of the traces and that an event e, of that execution trace, experiences a slowdown due to the variation. Furthermore, assume that e is not in the causal region of the release event $e_{\downarrow}$ of the variation. Our definition does not signal a* TA*, while intuitively one would expect a* TA*.*

However, given that the slowdown is due to the variation, some ordering must exist between $e_{\downarrow}$ and e, which has to be captured by a corresponding path in the ETDG *(Lemma 1). At least one arc along this path was removed according to Lemma 3. The rules of Inst. 7.5 referring to variations and resource switches are not applicable, since only a single variation occurred. The arc must have been removed due to a* timing gap*. This contradicts the hypothesis that e suffered a slowdown due to the variation (e was delayed by some other event), and the verdict of our definition must be correct.*

3. ***Slowdown:*** *Finally, assume that a (single) favorable variation is present in the input traces, that an event e suffered a slowdown due to the variation, and that e is in the causal region of the variation's release event $e_{\downarrow}$, but e does not exhibit an increase in relative time (Δ) for the favorable trace (i.e., with the favorable variation). Intuitively, a* TA *should be signaled, due to this* slowdown*.*

Having a single variation, and no resource switch, means that both traces are identical up to and including the acquisition events of the variation. Given that the acquisition events occur at the same instant, that the variation is favorable (δ), and that the relative time distance (Δ) is not larger in the favorable trace, it follows that also the absolute time of e is smaller in the favorable trace. This contradicts that e suffered a slowdown, and the verdict of our definition must be correct.

Traces that do not satisfy the three conditions in Def. 7.13 lead to a verdict that is coherent with the intuitive notion of TA*s. It remains to show that our definition is also coherent with this intuitive notion when it actually signals a* TA*.*

For this, assume that a (single) favorable variation is present in the input traces and that an event e exists that is both causal wrt. the variation and whose relative time distance increased in the favorable trace, but that did ***not*** *experience a slowdown. In terms of the intuitive notion, no* TA *should be signaled, while our definition clearly does.*

As before, we need to contradict the fact that e did ***not*** *experience an* intuitive *slowdown. For this, we can analyze the impact of the relative slowdown on e's timestamp:*

1. If the increase of the relative time distance (Δ) is larger than the amplitude of the variation (δ), its absolute time becomes larger in the favorable trace. It is difficult to attest the absence of a slowdown for e when both its relative and absolute times increase. This leads to a contradiction, and the verdict of our definition must consequently be correct.

2. The increase of the relative time distance (Δ) is not large enough and e occurs at the same time[a] or even earlier[b] in the favorable trace than in the other trace—which leads to a quite controversial situation, and the intuitive notion of TA*s is no longer sufficient to reach a conclusion.*

We argue that our definition still provides a sensible verdict for two reasons. First of all, we can find examples that reflect the same TA *pattern (see Sec. 8.2) with regard to some event e, with the only difference that in one example, e occurs earlier and in the other example, e occurs later (in terms of the absolute time). Since both examples exhibit the same pattern, the verdict should be the same*

for both examples—which is the case for our definition. Secondly, a strong link between the relative slowdown and causality exists (Lemma 4), which ensures that the accumulated delay up to e in the favorable trace is always greater than that of its corresponding event in the other trace, i.e., the OoO processor performs more work sequentially in the favorable trace between the variation and e. The increase in sequential *work reflects the intuitive notions of global slowdowns and thus* TA*s, even when the absolute time is not impacted.* ■

[a] For a concrete example, see Fig. 8.3.
[b] For a concrete example, see Fig. 8.4.

General Scenarios Having multiple variations, possibly combined with resource switches, goes beyond the intuitive notion of TAs and the capabilities of the existing definitions (cf. Ch. 6). We argue that our definition still provides sensible verdicts with regard to **a specific** variation in the input traces. The *causal region* of this specific variation allows us to clearly identify the events that are directly impacted by the variation from those that are not impacted at all, or from those that are impacted in addition by other variations. Our definition thus allows us to reason only about the events that are directly impacted—with correctness arguments similar to those for the case of a single variation (cf. Lemma 6).

Beyond individual variations, this leads to a new challenging problem: how can we classify TAs that are composed of possibly many variations and resource switches? Causal regions are an important step towards answering this question, which is out of the scope of this thesis and for now remains an open problem. We provide some illustrative examples in the next chapter.

7.4 . Summary: a Solid Groundwork for the Detection of TAs

In this chapter, we proposed a formalization of counter-intuitive TAs that addresses the issues caused by the existing definitions.

Our formalization allows accurate reasoning about multiple variations and the resource use of instructions. It is based on a specialized data structure, the Event Time-Dependence Graph, which is refined into a data structure that integrates the notion of causality, namely the Causality Graph. We sketched the open problem of the composition of variations (or resource switches).

We also instantiated our formalization on the OoO-pipeline template, with clear assumptions that allows for deriving a detection procedure of TAs for the concrete hardware model introduced in Ch. 5.

8 – DETECTION PROCEDURE

OUR detection procedure is code-specific, i.e., it performs the verification of the presence/absence of counter-intuitive TAs from a given input program only, namely for the combination of hardware and software (cf. Ch. 1). First, real-time systems execute a *specific* software on the target hardware, and, moreover, timing analyses are conducted for this software. Then, most of modern microarchitectures are subject to TAs, due to complex mechanisms acting as performance enhancers (cf. Ch. 1). Deeming a hardware intrinsically timing-anomalous could only deter from using it for real-time applications or, at best, guide for new designs (that would definitely discard some features), while a code-specific approach paves the way for efficient, localized (software or hardware) counter-measures.

Our definition of counter-intuitive TAs, sustained by the various instantiations in Ch. 7, constitutes a consistent TA-identification procedure for two input execution traces, from the same program trace executed on the representative out-of-order-pipeline hardware model (cf. Sec. 7.2.6). Our TA-detection procedure naturally consists in applying, with automatic tools, this identification procedure to a set of pairs of possible execution traces of the input program trace (i.e., from various initial hardware states).

Hereafter, we introduce the required adaptations to integrate a detection procedure based on our formal definition of TAs into the formal framework introduced in Ch. 5 (Sec. 8.1). Then, we interpret the results of the detection of TAs on short instruction sequences, in order to assess the theoretic capabilities of our procedure and the underlying formal definition of TAs (Sec. 8.2). Finally, we apply the detection procedure on standard benchmarks, showing that our work provides a solid basis for a tool support in the detection of TAs (Sec. 8.3).

8.1 . Adaptation of the Formal Framework

To apply our definition of counter-intuitive TAs, both the formal pipeline specification and the verification framework, introduced in Ch. 5, have to be adapted, wrt. Sec. 7.2.1 and, respectively, Sec. 7.2.2 to 7.2.5.

8.1.1 . Formal Specification

The full version of the TLA^+ specification accurately describes the parameterized OoO-microarchitecture case study (cf. Sec. 2.3). As mentioned above, it relies on a state variable, rob, to model the finite ROB through a sequence of instructions; it also relies on an additional state variable, $robHead$, pointing to the first instruction

in the rob sequence that actually resides in the ROB.[1] The TLA^+ specification thus allows us to derive pairs of execution traces from the target hardware model (cf. Inst. 7.1). However, we need to implement functions *Events* and *IDeps* in our specification (cf. Inst. 7.2). To do so, we rely on a state variable (with one copy per trace instance), $graph$, that keeps track of the events of the trace (in a similar way to $prog.exec$ in Ch. 5), namely the events of the ETDG (cf. Def. 7.5), as well as the RAW dependencies, in the form of source-destination instruction-index pairs where the destination instruction depends on the source instruction:

```
graph' = [ graph EXCEPT                                                          1
    !.nodes = [i ∈ 1 .. robHead − 1 ↦ graph.nodes[i]]                            2
              ∘ [j ∈ 1 .. Len(graph.nodes) − robHead + 1 ↦                       3
                LET i ≜ robHead − 1 + j IN                                       4
                [IFacq  ↦ graph.nodes[i].IFacq,                                  5
                IFrel   ↦ IF ∃ k ∈ (1 .. superscal) : _IF[k].instr = program[i]  6
                                                    ∧ _IF[k].baseLat = _IF[k].currLat  7
                           THEN currCycle + 1 ELSE graph.nodes[i].IFrel,         8
                ...]]                                                            9
              ∘ AppendRow(superscal),                                            10
    !.deps = LET dep(x) ≜ {entry ∈ robHead .. RobSelect(x) − 1 :                 11
                rob[entry].instr.r0 ∈ {x.r1, x.r2} \ {""}} IN                    12
             @ ∪ UNION ({{[source ↦ rob[x].instr.ind,                            13
                  dest ↦ _FU[k].instr.ind] : x ∈ dep(_FU[k].instr)} :            14
    k ∈ {j ∈ 1 .. N_FU : NotEmpty(_FU[j].instr) ∧ _FU[j].currLat = 1})           15
```

This state variable, which is a record composed of two fields ($nodes$ for the events and $deps$ for the dependencies), is updated according to the *observation* of the states variables that represent pipeline stages. The TLA^+ keyword EXCEPT (line 1), in association with the symbol !, allows both fields to be updated separately (from lines 2 and 10, respectively).

Events

Field $nodes$ is a sequence, where each element contains events related to the same instruction. Each of these elements is a nested record, where the various fields store the *timestamps* of the events that precisely refer to Inst. 7.2.[2] The update of the events preserves those related to instructions that are actually no more present in the ROB, i.e., that have committed and have thus left the pipeline (see $robHead - 1$ in line 2). The preserved sub-sequence (line 2) is composed with the updated sub-sequence of already observed instructions (lines 3-9), as well as with the sub-sequence of newly entered instructions (line 10), through operator $AppendRow(\_)$ (not detailed). The field for event $IFacq$ is unchanged

[1] Instructions are never removed from the rob variable, only the pointer ($robHead$) is updated.

[2] An additional field in the nested records identifies the instruction in question.

(line 5), since it is assigned its definitive value by $AppendRow(\_)$. The fields for the other events, for instance $IFrel$ (line 6), are updated by observing the related state variables, in this case $\_IF$. If one IF stage contains the instruction that has the same index in field $nodes$ as in the input $program$ (line 6), and moreover this instruction is about to complete the time required for fetching (line 7, cf. Sec. 5.1.2),[3] then the timestamp of the event corresponding to the release of IF by this instruction is updated (line 8), with the value of the *next* cycle (cf. Inst. 7.2). Otherwise, the timestamp is unchanged (line 8). The same reasoning applies for all other events (line 9).

Data Dependencies

Field $deps$ is an (unordered) set that contains pairs of dependent instructions. A RAW dependency may be detected for instruction x among all the instructions actually present in the ROB (see $robHead$) and preceding in the ROB—operator $RobSelect(x)$ selects the index in state variable rob where x is stored (line 11). A dependency is detected when such an instruction has a destination register (field $r0$) that fits one of the source registers (fields $r1$ and $r2$, if they are not empty) of x (line 12). To reduce the size of the ETDG, which partially relies on data dependencies (cf. Inst. 7.3), not all static dependencies that exist in the program are captured, but only the dependencies within the dynamic window where a previous instruction may have an impact on the dependent instruction (in the same vein as the exclusion of causality rules that would cause unnecessary arcs by transitivity).[4] We thus focus specifically on instructions that have just entered a FU (line 15)—$NotEmpty(\_)$ ensures that an instruction is not the conventional empty instruction. For any such FU (indexed by k), we specify the set of $source$-$dest$ records (lines 13-14) that have the instruction in the FU as the destination instruction and an instruction on which the destination instruction depends as the source instruction. Note that this set is empty if there is no dependency. Then, we specify the UNION (line 13) of the sets (of sets) related to distinct FU indexes k. Finally, we update field $deps$ through the union ($\cup$) of this flattened set in turn with the current value of the field (line 13)—the symbol @ is a syntactic sugar for the current value of the field under consideration (i.e., corresponding to the last "!"). Both functions *Events* and *IDeps* are thus implemented.

8.1.2 . Property for the Absence of TAs

In order to capture causal relationships and accurately detect TAs, our definition relies on graphs that have to be explored in specific ways. In particular, several calculations are necessary to establish causal regions in a trace (Def. 7.12), in a *single* cycle, and the number of operations is not known in advance, since it depends

[3] The instruction is not necessarily about to leave the stage—stalling may occur (cf. Sec. 7.2.1).

[4] Note, however, that this is a different issue from the removal of arcs according to causality (cf. Ch. 7), since we also do capture data dependencies that may *not* have a direct impact on the dependent instructions.

on the structure of the trace. Consequently, expressing the property stating the absence of TAs under our definition is difficult through temporal logic. TAs are complex properties, not only since they manifest at the level of two traces (which can be easily resolved, see Sec. 5.3.1), but also since they refer to distinct, arbitrarily distant states of a trace to express local and global variations. The properties that we proposed in Ch. 6 for detecting TAs under the previous definitions are quite simple to express in the form of temporal-logic invariants (cf. Sec. 3.2.1), through a few state variables (or fields) that actually keep track of the history of traces. Nevertheless, even for these properties, the most complex (and fine-grained) case ($TALoc$) is not trivial and relies on several auxiliary state variables.

We have progressively established our definition in the purpose of *identifying* TAs (see Sec. 7.2). This inherently leads to an *algorithmic* description of the identification procedure, as we sketched in Sec. 7.2.6.

Black-Box TLA$^+$ Property

A convenient feature of the model checker TLC is the possibility for the user to *override* TLA$^+$ modules [48], thus to encode algorithms. The operators defined in overridden modules must only be syntactically correct, whereas their semantics is transferred to the (Java) code integrated in the sources of the model checker—the TLA$^+$ operators act as black boxes. We thus define an operator to provide the result of our identification procedure of TAs on the current traces from both instances:

$$HasTA(g1, g2, cycle, onlyCom) \triangleq \text{FALSE}$$

This operator is overridden by Algorithm 8.1, encoded in a general-purpose programming language.[5] This algorithm is at the heart of our detection procedure, since it formalizes the identification procedure sketched in Sec. 7.2.6. Then, we can simply verify the absence of counter-intuitive TAs in each cycle (i.e., each state of our dual-execution specification), with the following invariant:[6]

$$\neg HasTA(graph, graph2, currCycle, onlyCom)$$

where $graph$ and $graph2$ are both copies of state variable $graph$ (cf. Sec. 8.1.1), which allow building the graphs (ETDGs/CGs) of both traces. Our detection procedure thus relies on two exploration phases: the engine of TLC explores the possible variations to build pairs of traces on the fly (first phase), while the very identification of TAs is performed through the implementation of Algorithm 8.1 that explores the specific (ETDG/CG) graphs, for two fixed input traces (second phase).

Algorithm 8.1: TA-identification procedure for a pair of traces from the OoO-pipeline model.

Input: $\mathit{Events}(\alpha)$, $\mathit{IDeps}(\alpha)$, $\mathit{Events}(\beta)$, $\mathit{IDeps}(\beta)$; ▷ Def. 7.2-7.3 / Inst. 7.2: arguments $g1, g2$ of $HasTA$ (cf. Sec. 8.1.1)
Output: Presence of TAs ; ▷ Overridden Boolean-valued operator $HasTA$
Result: ETDGs G_α, G_β, CGs C_α, C_β, set of TAs $\{(e_\downarrow, e)\}$; ▷ Graphical results

1 **foreach** $\tau \in \{\alpha, \beta\}$ **do**
2 $tdeps_\tau = \mathit{TDeps}(\mathit{Events}(\tau), \mathit{IDeps}(\tau))$; ▷ Def. 7.4 / Inst. 7.3
3 $G_\tau = (\mathit{Events}(\tau), tdeps_\tau)$; ▷ ETDG (Def. 7.5)
4 **foreach** $\tau \in \{\alpha, \beta\}$ **do**
5 $variations_\tau = \emptyset$;
6 $causalArcs_\tau = tdeps_\tau$;
7 **foreach** $a = [e_1 = (i_1, r_1, t_1) \xrightarrow{w} e_2 = (i_2, r_2, t_2)] \in tdeps_\tau$ **do**
8 **if** $a \notin \mathit{Causality}(G_\tau, G_{\bigcup[\{\alpha,\beta\}\setminus\{\tau\}]})$ ▷ $\bigcup$ simply selects the *other* trace. **then**
9 $causalArcs_\tau = causalArcs_\tau \setminus \{a\}$; ▷ Def. 7.10 / Inst. 7.5
10 **if** $(i_1 = i_2 = i) \wedge \exists u, (r_1 =\uparrow u \wedge r_2 =\downarrow u)$ ▷ Resource use (matching acquisition/release events) **then**
11 $(i, r'_1, t'_1) = \mathit{CospEvent}(e_1)$; ▷ Def. 7.7 / Inst. 7.4
12 $(i, r'_2, t'_2) = \mathit{CospEvent}(e_2)$;
13 $\delta = t_2 - t_1, \quad \delta' = t'_2 - t'_1$; ▷ Latency (Def. 7.6)
14 **if** $\delta \neq \delta'$ ▷ Variation (Def. 7.8) **then**
15 $variations_\tau = variations_\tau \cup \{(e_1, e_2, \delta, \delta')\}$
16 $C_\tau = (\mathcal{N}_\tau, causalArcs_\tau)$; ▷ Causality graph (Def. 7.11)
17 $TAs = \emptyset$;
18 **foreach** $\tau \in \{\alpha, \beta\}$ **do**
19 **foreach** $(e_\uparrow, e_\downarrow, \delta, \delta') \in variations_\tau$ **do**
20 **if** $\delta < \delta'$ ▷ *Favorable* variation (Def. 7.8) **then**
21 **foreach** $e \in \mathcal{N}_{C_\tau(e_\downarrow)}$ ▷ Causal region (Def.7.12) **do**
22 **if** $\Delta(e_\downarrow, e) > \Delta(\mathit{CospEvent}(e_\downarrow), \mathit{CospEvent}(e))$ ▷ TA (Def. 7.13) **then**
23 $TAs = TAs \cup \{(e_\downarrow, e)\}$;

Algorithmic Identification Procedure

Algorithm 8.1 is a procedure that permits finding the TAs triggered by either trace at any event, based on the definitions introduced in Sec. 7.2. Its implementation relies on the instantiations introduced in Sec. 7.2, through a straightforward, sequential encoding of the rules described in the instantiations. The algorithm takes as input the two traces for which we wish to identify TAs and outputs a Boolean indicating the existence of TAs—triggered by events of any of both input traces wrt. the other one. It also produces the set of discovered TAs, as well as the computed graphs for each trace. The flow of the algorithm is similar to the description provided in Ch. 7, allowing for identifying TAs in both directions (where traces switch positions). However, the variations and the causal arcs in an ETDG are identified in a single step—a single exploration of an ETDG permits the identification of variations and the progressive construction of the CG—and, similarly, the TAs triggered by each favorable variation are identified incrementally as the causal region of the variation is explored.

1. The first step consists in determining the timing dependencies of events, for each trace (line 2). From these timing dependencies, we directly build the ETDG of each trace (line 3).

2. Once both ETDGs are built, we can derive the variations and the causal arcs of each. We successively explore both of them (line 7). The causal arcs are derived by the elimination (cf. Inst. 7.5) of the other arcs (lines 6 and 8-9). If an arc connects two matching acquisition/release events (line 10, cf. Def. 7.6), the arc represents a resource use and we can compute the related *latency*. Note that the arcs that represent a variation in the use of a resource are *not* causal (cf. Rule 2 Inst. 7.5). From the corresponding events (lines 11-12, cf. Inst. 7.4), we compute the latencies in both ETDGs (line 13) and we thus identify (line 14) and save (line 15) the *variations*. At the end of this step, the causality graphs are fully determined (line 16) and, hence, we can explore causal regions in these graphs.

3. For each trace, we analyze the identified variations and we consider only the *favorable variations* (line 20). We compute the *causal region* of each favorable variation (from the resource release) in the CG of the trace (line 21). For each event node e of this region, we compare the *relative time distance* between this node and the release event involved in the variation, obtained via the Δ operator (cf. Sec. 7.2.5), with the corresponding time distance in the other trace (line 22). A TA is signaled when the favorable variation is associated with a larger relative time distance, thus indicating a (temporary) slowdown (line 23).

[5] The sources are available in the aforementioned repository.

[6] $currCycle$ is supplied for debugging/logging purposes and $onlyCom$ can be used to restrict the events that may trigger TAs to commit events.

8.2 . Interpretation of our Detection Procedure on Short Sequences

The detection procedure established in the previous section allows us to query the model checker for TAs: **Does a given instruction sequence exhibit TAs? May TAs disappear when restraining the initial hardware state?** The subsequent illustrations and examples are all obtained using this tool. The input sequences are based on the short example found in the literature (cf. Sec. 2.3.3) and adapted in order to highlight interesting features of our definition (e.g., through variations of the parameters of the OoO model, in particular $superscal$ and N_{FU}). For better readability, we use only a compact trace representation similar to Fig. 7.1 (which integrates the main information of the CG of the trace).

8.2.1 . Basic Variation Cases

We start with a series of simple examples, by opposing the results obtained using our definition with previous work and the intuitive notion of TAs. We show that these examples, entailing surprising results for the major formal definitions, are correctly handled by ours.

Unrelated Variations

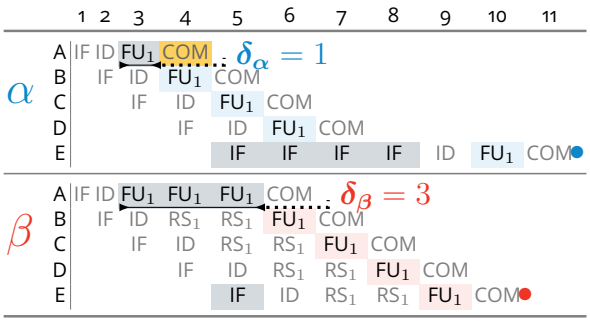


Figure 8.1: Illustration of the separation of unrelated variations.

Fig. 8.1 shows two execution traces on an in-order configuration of our OoO model ($superscal = 1$, $N_{\mathsf{FU}} = 1$). The traces contain two variations, where one is favorable for trace α and the other one is favorable for trace β. The other two traces (i.e., resulting from the other two combinations of the variations) might have been pruned (see Sec. 7.1.1). If we try to apply the definition by Reineke et al. [22] to α and β, the situation of the reference example of Ch. 7 is reversed. We can identify a local-worst-case trace, namely trace β. The variation in FU_1 is a local worst case for β: $\alpha_{\mathsf{pre}} = \beta_{\mathsf{pre}} = (\alpha_1, \alpha_2)$ and $1 = \left|\alpha_{|\mathrm{EX}(A)}\right| < \left|\beta_{|\mathrm{EX}(A)}\right| = 3$. Since both traces have already diverged in cycle 5, when the second variation occurs, and the traces consequently do not share the same prefix, no local worst case is identified for this variation and only the first observed variation serves as a basis to define the local-worst-case trace. However, the first variation is irrelevant

wrt. the global execution time. The scheduling on FU_1 is exactly the same in both traces, and the global execution time depends on the larger latency among both variations, namely the latency of the second one in this case. More generally, while the intuitive definition clearly leads to the absence of TAs, all existing definitions, surprisingly, state a TA (cf. Sec. 6.1.2).

Our approach splits the favorable trace α into independent parts. The causal region of the first variation in α is limited to the commit event of the first instruction, $(A, \mathrm{COM}, 4)$, since the successive instructions do not have data dependencies and they do not experience a resource contention. The relative time distances of the COM event wrt. the variation in both traces is constant (1 cycle). Hence, our definition correctly states the absence of TAs.

TA with a Limited Impact

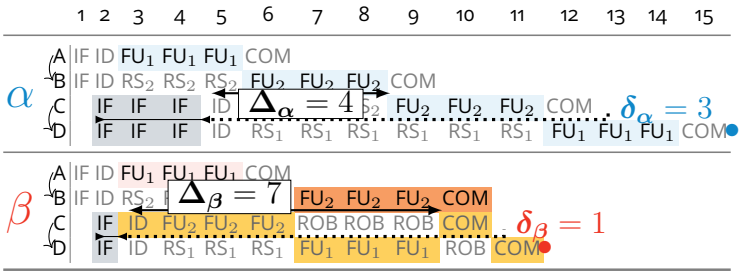


Figure 8.2: Example showing that TAs may be limited in scope.

Since the causal region allows precisely capturing the scope of a variation, we can also detect TAs that have a limited impact on the execution. In the example from Fig. 8.2, we observe that the execution of instruction B in FU_2 and its completion in COM occur later in the favorable trace β. However, the trace with the favorable variation has the shorter global execution time. The definition by Reineke et al. [22] does not state a TA, since α is a local-worst-case trace and is longer. Note that the definition by Gebhard [24] for instance would signal a TA caused by instruction B (cf. Sec. 6.1.2): this instruction has a shorter latency $\gamma(\eta_\alpha, B) = 3$ cycles in α vs. $\gamma(\eta_\beta, B) = 4$, while the global execution time is larger in α ($\Gamma(\eta_\alpha, D) = 15 > \Gamma(\eta_\beta, D) = 11$). However, it is clear that B does not cause the TA.

In our case, the TA is clearly attributed to the variation at instruction C, which blocks instruction B due to a resource contention on FU_2 in trace β. Such effects are generally not captured in previous work. **Our definition also captures the fact that a TA has an effect on a limited scope.** In this example, instruction D neither experiences an absolute nor a relative slowdown wrt. the variation. Thus no TA is signaled here.

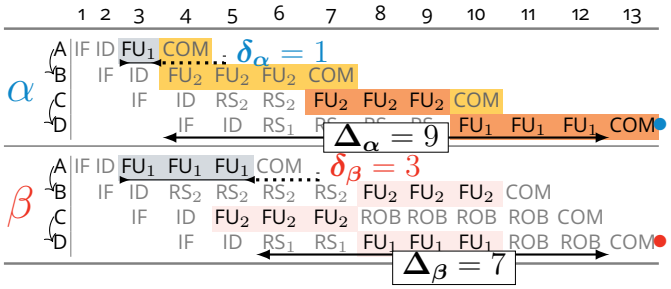


Figure 8.3: Example showing a TA pattern that does not impact absolute time.

Identification of TA Patterns

Fig. 8.3 shows a variant of the common example (cf. Sec. 2.3.3), executing on a constrained OoO model ($superscal = 1$, $N_{\mathrm{FU}} = 2$). All existing definitions signal the absence of TAs due to the identical global execution time (cf. Sec. 6.1.2). Yet, the global scheduling pattern characterized by the use of FUs is the same as in the common example, which clearly exhibits a TA. The situation is similar to the TA detection on traces α/α' vs. β' in the reference example of Ch. 7, but this time the traces exhibit a single variation.

Our definition is based on the precise identification of relevant uses of resources, which leads to the detection of this TA pattern as of the acquisition of FU_2 by C, i.e., the event $(C, \uparrow \mathrm{FU}_2, 7)$. This resource can indeed be used even before the end of the variation in β, i.e., the corresponding relative time distance in β is negative. Moreover, the TA persists up to the end of the execution, although the global execution time is the same in both traces. This may surprise, but our definition relies on the *relative* time distance from the resource release of the variation, **in order to capture actual *slowdowns*** instead of the absolute execution time. The commit of D in α does suffer a slowdown in α wrt. β due to the sequential execution of instructions B, C, and D.

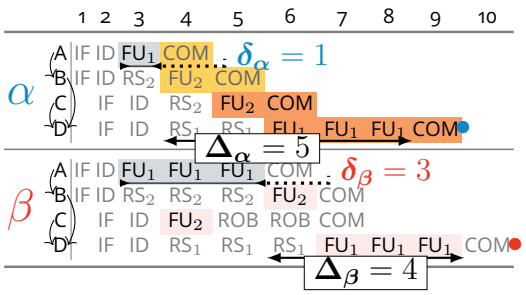


Figure 8.4: A TA pattern in contradiction with the absolute time.

Fig. 8.4 shows that our definition does capture TA patterns, relying on relative time, even when the favorable trace is associated with a smaller absolute time. In this example with a different configuration ($superscal = 2$), an additional data dependency, and shorter default latencies in FUs, the commit events $(C, \mathrm{COM}, 6)$

and $(D, \mathrm{COM}, 9)$ in the favorable trace α, notably, have smaller timestamps than the corresponding events in trace β (7 and 10, respectively). However, TAs are triggered, due to the larger relative time distances from the variation (see the figure for the commit events of D in both traces). Here again, the stated TAs consistently reflect the slowdown in α due to the sequential execution of instructions B, C, and D.

8.2.2 . General Scenarios

Next, we consider more complex examples that exhibit several variations or even FU switches. These considerations are overlooked by all of the existing definitions, though these definitions do not exclude them from their hypotheses. Our approach consistently handles variations by identifying their individual impact, through causal regions. Within a causal region, the *relative* time distance enables us to focus on the effects of the last variation, excluding any resource switch. This brings up the problem of the *composition* of multiple variations/switches. If none of the variations taken independently triggers a TA, we might intuitively suspect that the composition of the variations does not exhibit a TA. However, the composition of variations/switches and TAs in general is an open problem. We illustrate this in the following examples.

Serial Composition

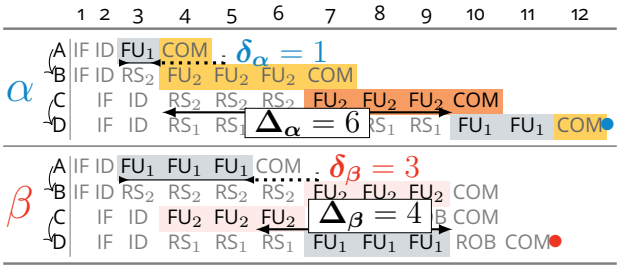


Figure 8.5: Composition of two variations with a clear verdict.

In Fig. 8.5, we consider the slightly modified common example (cf. Sec. 2.3.3), where instruction D also has a variation. Both variations occur one after the other in one trace and, moreover, the second variation is time-dependent on the first. Differently stated, the causal region of the first variation contains the acquisition event of the second variation. Rules 2 and 3 of Inst. 7.5 ensure that the causal region ends with this event, similarly to the cases where there is no timing dependency (cf. Fig. 8.1) or there is a timing gap, since the remainder of the execution behavior does not depend only on the first variation. Consequently, we do detect the same TA as previously (cf. Sec. 7.1), triggered by the first variation, up to the commit of C.

Since both variations are time-dependent and favorable for the same trace, this situation leads to a *serial composition*. The second variation only reduces

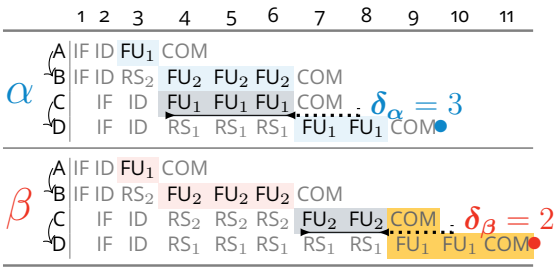


Figure 8.6: Example highlighting the impact of resource switches independently from the actual (latency) variation.

the global execution time in α; it does not entail any particular timing effect nor globally prevent the TA from occurring. We thus could extend the causality region of the first variation in order to compute the relative time distance up to the end of the trace and state a global TA. Similarly, we intuitively suspect that the serial composition of two TAs remains a global TA, since in this case the second variation even exacerbates the already observed slowdown. However, as the subsequent examples show, the analysis of a composition of TAs is complicated in general. **Our definition provides a starting point to tackle the problem of *compositions* in future work.**

Combined Variation with FU Switch

Through the example in Fig. 8.6, we show that we consistently handle variations with resource switches. We consider one variation combined with a FU switch, namely C uses different FUs in each trace. We identify one favorable variation and our definition states no TA, since the relative time distances are the same in both traces. This statement is consistent: the use of—whatever—FU by C implies no particular scheduling effect for the rest of the trace and we observe exactly the same tail in both traces from the variation onward. The difference in the global scheduling only results from the FU switch that delays the *acquisition* of the FU in β. We suspect the delay thus introduced until a resource is available to be similar to amplification effects (cf. Table 2.2).

Composition with a Series of TAs

Let us now consider the example in Fig. 8.7, in which instructions A and C exhibit variations and instruction C, in addition, switches its FU. The definition by Reineke et al. [22] does not signal a TA, since the first variation is favorable for the shorter trace (α). Gebhard's definition [24] would attribute a TA to instruction D (latency of 2 cycles in α vs. 1 in β).

For our definition, two favorable variations are identified, one for instruction A in α and a second for instruction C in β. The former variation alone does not trigger any TA, as indicated in the figure (■). The latter triggers TAs by itself (■), in particular for D's commit (see Δ_α and Δ_β). This allows us to state that

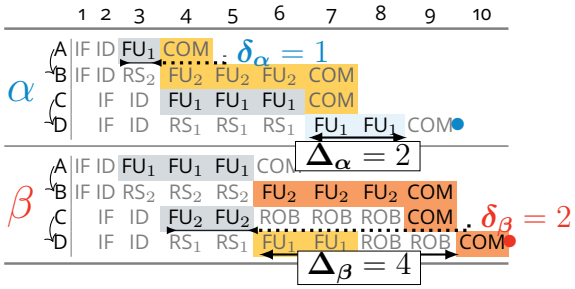


Figure 8.7: Illustration of cumulative effects of multiple variations.

TAs occur in this example.

The characterization of the global timing behavior is tricky, due to the interaction between the *opposing* variations. Let us focus on the favorable variation for α, which clearly does not trigger TAs in α. Note that we observe an increase in the relative time distance wrt. the commit of instruction D (5 cycles in α, vs. 4 in β)—a slowdown. This event is not in the causal region of the variation, since it is exclusively delayed by C. If we focus on trace β, we observe that the resource switch of C imposes a delay on B due to contention on FU_2. However, the increase of the use of FU_1 by A is crucial in determining the execution order between B and C. Without this increase, the example would result in the same trace as Fig. 8.6. The variation on A thus also plays a role in the appearance of the TAs visible in β. Due to the independence of these choices (Inst. 7.1), causality is excluded though. This shows that the problem of composition is complex and needs to be investigated further—notably, considering more realistic processor implementations, where these choices will necessarily expose causal relationships.

Composition with Mutual TAs

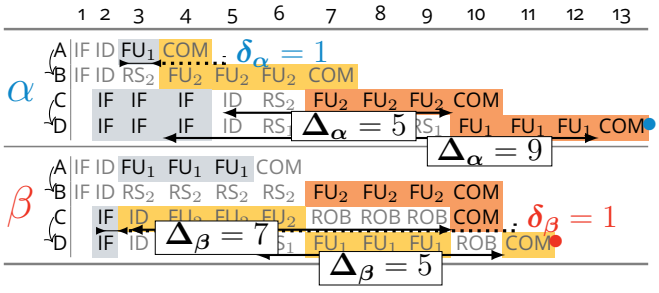


Figure 8.8: Composition triggering mutual TAs in both traces.

Now, let us consider Fig. 8.8, a combination of the common example (cf. Sec. 2.3.3) and the traces from Fig. 8.2, i.e., a variation in the use of IF by C and D is added. The global execution times and scheduling effects in the resulting traces are the same as in the common example. This is due to the fact that the use of IF by C, though longer, is still too short for its release to delay any relevant event in the trace and thus to have some effect on the execution. Intuitively, there

thus should be at least the same TAs as in the common example. The definition by Reineke et al. [22] does not signal a TA, since the first variation is favorable for β, which is also the shorter trace. The definition by Gebhard [24] signals a TA for instruction B ($\gamma(\eta_\alpha, B) = 3 < \gamma(\eta_\beta, B) = 4$ and $\Gamma(\eta_\alpha, D) = 13 > \Gamma(\eta_\beta, D) = 11$).

Our definition remains consistent, identifying the same TAs for α as in the common example. Moreover, TAs are identified due to the favorable variation of IF for C in β, which are consistent with those identified in Sec. 8.2.1. The difference is that the commits of B and C occur earlier in α for this example, which explains the TA for instruction C in β. **Our definition is thus able to clearly separate the effects of those mutual TAs.**

8.3 . Detection of Timing Anomalies on Benchmarks

In this section, we present a partial evaluation of the capability of our procedure to detect TAs on standard benchmarks executed on our OoO-hardware model, specifically the TACLe benchmark collection [105]. This benchmark collection is commonly used as a case study in the real-time community [25, 97, 106].

8.3.1 . Strategy and Heuristics

Our detection procedure accurately captures all counter-intuitive-TA effects that may arise in the scope of the causal region of any (favorable) variation. It can thus theoretically detect any TA that may occur during the execution of a program. However, in practice, we need to specify a certain set of possible variations that must be explored by the model checker and we must ensure both the feasibility of these variations and the correctness of the verdicts about TAs.

Variation sources

On the one hand, we assume that variations result from the behavior of the cache, but we do not explicitly model the cache: the possible variations must be predetermined; on the other hand, a too wide set of variations (e.g., an over-approximation) entails state explosion and might hinder the detection of TAs. Consequently, we need to rely on heuristics to derive a reasonable set of variations, targeting specific variation points. We will see that this often allows highlighting TAs, but, otherwise, the specified set could be refined to account for more variations. The heuristics could rely on preliminary static (instruction- and data-) cache analyses, to detect the TAs that may occur without perturbations. However, as a first step, we set up, for our evaluation, a particular heuristic ($\mathtt{H_{var}}$) that targets the data-cache misses as in the traditional pattern (see Fig. 7.2).[7] In this pattern, (RAW) data dependencies constitute a constraint that enforces a *sequential* execution in certain traces and thus may entail TAs. With this heuristic, only the *memory*

[7] Note that we thus exclude here, as a first step, the instruction-cache misses that may entail TAs, e.g., as in Fig. 8.2.

instructions (i.e., load/store operations, which may access memory) on which a subsequent instruction is *dependent* are allowed to exhibit a variation, in the FU. The resulting number of variations depends on the application and the number of instructions in the input program. Note that this heuristic is not part of the TLA^+ specification; the possible variations are still explicitly supplied as input. For convenience, we adapt the specification with a separate input parameter, $mayDMiss$, that specifies the set of (indexes of) the instructions that may exhibit variations in the FUs.[8] The total required latency in a FU (cf. Sec. 5.1.2) is thus assigned as follows:

$$\text{IF } NxtFU(i).ind \in mayDMiss \text{ THEN } \{latency,\ missLat\} \text{ ELSE } \{latency\}$$

where $NxtFU(i)$ returns the instruction that is to be scheduled in the FU (cf. Sec. 5.1.2), $latency$ is a default latency, and $missLat$ is the input latency that represents a cache miss.

Verification Instants

The detection problem cannot either be decomposed into sub-problems in a straightforward manner, since TAs might be triggered at any event of causal regions and causal regions may only be bounded according to the length of the whole trace. The copies of the $graph$ state variable that store the events and the dependencies of both traces (see Sec. 8.1.1) keep track of an increasing, cumulative amount of information in each cycle (without any loss). Ideally, causal regions could be exploited to reset these variables and decompose the problem on the fly by considering only sub-graphs where at least one variation determines the timing of some events for each represented timestamp. Hereafter, we do not exploit causality but, as a first step to mitigate the repeated computations on the same parts of the graphs in successive states, we set up a second heuristic ($\mathtt{H_{check}}$). We check for TAs only in specific states, when predicate $CheckTA$ holds. The invariant stating the absence of TAs (cf. Sec. 8.1.2) thus becomes:

$$NoTA(onlyCom) \triangleq CheckTA \implies \quad (1)$$
$$\neg HasTA(graph,\ graph2,\ currCycle,\ onlyCom) \quad (2)$$

Under $\mathtt{H_{check}}$, predicate $CheckTA$ holds only when a number of instructions—in our case, S_{ROB}, the size of the ROB—has fully executed since the last state where the TA detection was performed (or since the initial state), or when the whole program has executed. The final verdict does not depend on this predicate (nor on the considered number of instructions), since the graphs may only grow in successive states where $CheckTA$ holds. This predicate allows a trade-off between the fast identification of counterexamples (with early calls to the identification procedure through $\neg HasTA$, potentially in each cycle) and the elimination of repetitive computations (which can be achieved by checking for TAs only once

[8] In this version of the specification, there is a direct match from instruction types to one operable FU, with specified default and miss latencies in the FUs.

both execution traces are complete). The latter possibility (checking for TAs only when both traces are complete) forms the alternative heuristic H'$_{\texttt{check}}$.

$$
\begin{array}{ll}
GraphBound \triangleq Min(\{robHead,\ robHead2\}) - 1 & 1\\
CheckTA \triangleq & 2\\
\quad \lor\ GraphBound - lastBound = S_{\mathrm{ROB}} & 3\\
\quad \lor\ \land\ Len(rob) > 0 & 4\\
\qquad \land\ rob[Len(rob)].instr = program[Len(program)] \land rob[Len(rob)].done & 5\\
\qquad \land\ graph.nodes[Len(graph.nodes)].COM \neq 0 & 6\\
\qquad \land\ \text{Similar for trace 2 (rob2, graph2)} & 7
\end{array}
$$

Operator $GraphBound$ computes the ROB index of the last instruction that has left the pipeline in both instances (line 1)—recall that $robHead$ points to the oldest instruction still in the ROB. State variable $lastBound$, initialized to 0, is updated with the value of $GraphBound$ every time that $CheckTA$ is TRUE and thus we check for TAs. Consequently, the first disjunct (line 3) is satisfied when the fixed number of instructions has executed. The second disjunct (lines 4 to 7) is satisfied only once, when the whole program has executed. The last instruction currently in the ROB (of the first instance) must be the last instruction of the input program and this instruction must have fully executed (line 5). Moreover, to ensure that this disjunct is satisfied only once, we add the condition that the commit event has not occurred yet (line 6). The condition line 4 is required not to entail errors in the subsequent conjuncts when the ROB is empty. We do the same for the second instance (line 7). Note that H'$_{\texttt{check}}$ is simply got by removing the first disjunct (line 3).

Symmetry in the Dual Specification

Finally, the two copies of the pipeline specification for reasoning on two traces may result in the exactly same execution. Moreover, contrary to the verification strategy based on previous definitions (cf. Sec. 5.3.2), we do not need here to explore pairs where both traces merely switch positions while exploring the variations (with TLC), since our identification procedure explicitly checks for favorable variations, and then TAs, for both traces (Algorithm 8.1). We can thus remove the symmetry between both execution instances, in order to get an `asym` version of the specification where the above condition for the assignment to $missLat$ becomes:

$$
\begin{array}{ll}
\text{IF}\ \land\ NxtFU(i).ind \in mayDMiss & 1\\
\quad \land\ exec\_inst = 2 \implies NxtFU(i).ind \neq Min(mayDMiss) & 2
\end{array}
$$

The second conjunct (line 2) removes one arbitrary variation in the second instance (e.g., the variation for the instruction with the smallest index among those that may show variations). Note that a single variation can be removed, to allow for all possible combinations of variations in both traces.

It may also be helpful to consider instead heuristic `asym'`, in which the trace of a single instance may exhibit cache misses (according to $mayDMiss$). In this way, we (temporarily) prevent compositions with opposite variations to occur and we

can focus on isolated traditional patterns of TAs. The compositions that can still manifest in this case are easier to interpret, since the favorable variations always occur in the same trace (see Sec. 8.2.2). Heuristic `asym'` is derived from the `asym` version of the specification by substituting the following line for above line 2:

$\wedge\ exec\_inst = 2$

8.3.2 . Workflow

In TACLe benchmarks, all input data are part of the C source codes and all flow constraints are incorporated into the code [105]; hence, the benchmarks are single-path programs, of which any execution thus corresponds to a single program trace, i.e., a sequence of instructions. To apply our detection procedure to benchmarks, we need to format this sequence of instructions to fit our program representation, namely the $program$ input parameter.

Fig. 8.9 represents the workflow for applying our detection procedure to the benchmarks, from an input C source code to the related values of the formatted input parameters for our TLA$^+$ specification. First, we target RISC-V microarchitectures, such as the BOOM core (cf. Sec. 2.3). We thus rely on the RISC-V `gcc` (cross) compiler to produce an executable file from the C source code, the execution of which always generates a single trace. Then, we use a simulator, `gem5`, to simulate the functional behavior of the processor under this executable and produce the execution trace. We thus get a *sequence* of instructions, from which, moreover, we automatically extract the instruction class (e.g., ALU or memory read) and the operands (i.e., in particular, registers). Besides the execution of the instructions of the input program, this sequence of instructions contains initialization instructions for the target microarchitecture. To focus actually on the benchmark under consideration, we retain the first instruction of the main function of the input code as the starting point for the input $program$ of our specification. To do so, we also disassemble the `.text` section of the obtained executable, and we search for the address of this first instruction. We use it to parse the simulation trace, from this instruction and with N_{instr} (supplied as an input of the workflow) instructions in total. We extract the relevant information of each instruction and we store it, in the purpose of generating the $program$ input parameter, in the appropriate format (cf. Sec. 7.2.1) for our specification.

We also remarked that $\mathtt{H}_{\mathtt{var}}$ (as well as any other potential heuristic for variations) is not part of the specification (see Sec. 8.3.1). We thus need to determine the possible variations, according to the considered heuristic, within the setting-up workflow. To apply $\mathtt{H}_{\mathtt{var}}$, we use a circular buffer that helps us identify the instructions that show (RAW) dependencies on a memory operation within the range of previous instructions determined by the maximal number S_{ROB} of instructions in the ROB (i.e., the number of instructions that can interact during execution, see Sec. 8.3.1). We use the derived list of instruction indexes directly as the value of parameter $mayDMiss$.

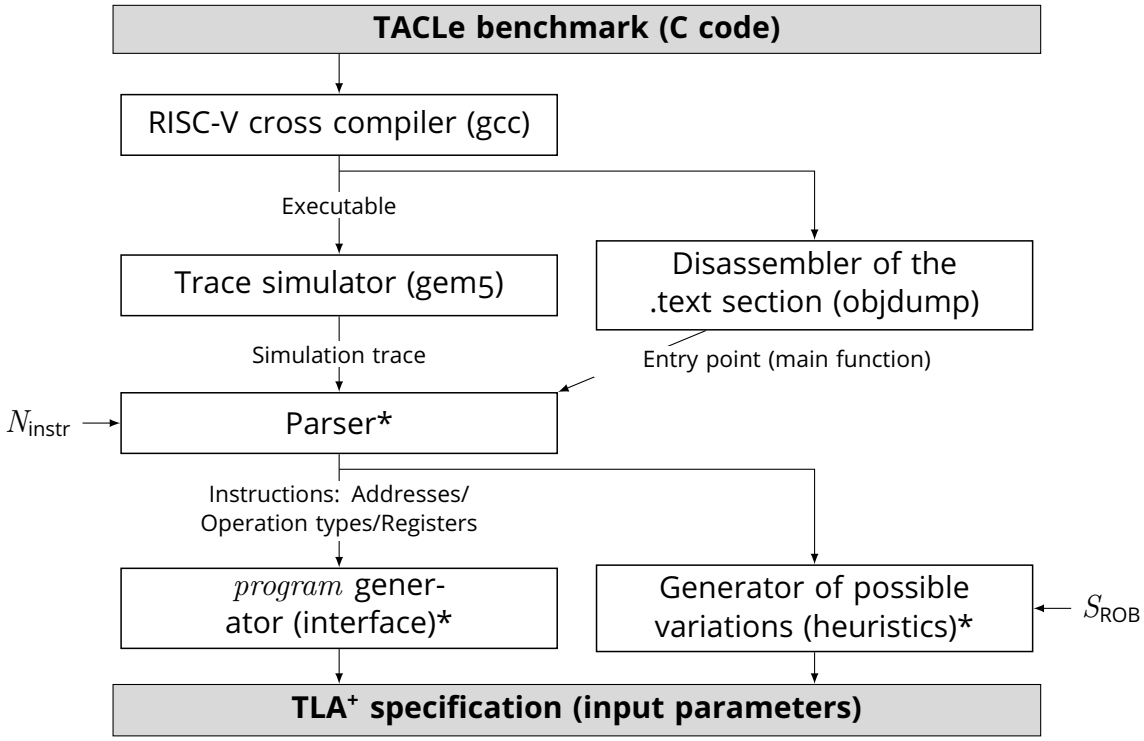


Figure 8.9: The setting-up workflow of input parameters transforms the trace encoded in a TACLe [105] benchmark (▭) into a suitable input $program$ for our TLA$^+$ pipeline specification (▭), accompanied with a set of possible variations derived from heuristics. It relies on standard tools (▭) and on personal scripts (▭*) to produce intermediate information (→), from supplied parameters (N_{instr} and S_{ROB}).

8.3.3 . Experimental Results

Hereafter, we report experimental results obtained from several TACLe benchmarks (column *benchmark*). Our assessment is not systematic but aims at showing that our formal definition can have a *practical* implementation. In this section, we fix the sizes of the buffers ($S_{\text{RS}} = 12$ in the TLA$^+$ specification and $S_{\text{ROB}} = 12$ in the specification and in the setting-up script, see Fig. 8.9).[9]

Verification Configurations

The numbers N_{FU} of FUs are specified according to the amount of distinct instruction types in the benchmarks, and the related default latencies are also fixed (e.g., 1 cycle for memory operations—cache hits—, 1 cycle for integer operations, except 4 cycles for integer division). The data-cache-miss latency is also fixed to be of 10 cycles. Besides, we always rely on $NoTA$ under the form $NoTA(\text{FALSE})$

[9]These values are chosen for illustrative purposes but are of the same order of magnitude as in other experiments, e.g., about WCET in OoO pipelines [35].

(not restricting reported TAs to COM events). However, we allow for variations in parameter $superscal$ (in the specification), as well as in parameter N_{instr} (in the setting-up script)—see the respective columns in the table. The remaining TLA^+ input parameters, the input $program$ and the possible variations $mayDMiss$ (whose cardinality is reported in the table), are generated for each benchmark (cf. Sec. 8.3.2). The latter parameter is always derived from heuristic $\texttt{H}_{\texttt{var}}$ (cf. Sec. 8.3.1). The number N_{instr} of considered instructions in the benchmarks may be progressively increased, in particular to reduce the state space by starting with a limited set of variations $mayDMiss$ from $\texttt{H}_{\texttt{var}}$ and thus to get quick results about the detection of TAs.

Interpretation of Results

Table 8.1 reports the results for the various verification configurations: the verdict from the model checker (column *result*: TRUE if the property is verified, or *cex* if a counterexample is found), the runtime needed to check the property indicated by column *invariant* (column *time*), the diameter (column *diam.*, i.e., the number of explored transitions), and the total number of states found by the model checker up to its verdict (column *states found*). When the property is not verified, the diameter corresponds to the way that the model checker searches the state space (in our case, the smallest counterexample, since we perform a breadth-first search). When the property is verified, the diameter is the maximal number of states, thus of clock cycles, required to fully execute the program. It must be (and it is indeed in Table 8.1) larger than $N_{\text{instr}}/superscal$ (cf. Ch. 1).

The first two configurations perform the verification from a trivial, valid invariant (column *invariant*), i.e., TRUE, on an arbitrary benchmark. In this way, we focus on the first exploration phase of our verification strategy (cf. Sec. 8.1.2) and we can easily check the impact of `asym` (cf. Sec. 8.3.1): the gain in verification runtime is not relevant for small runtimes (which include the initialization of the model checker), but, by comparing configurations (1) and (2), we can observe that the state space (penultimate column) is indeed divided by more than two. `asym` allows us to remove the duplicate scenarios where both instances switch their roles (thus about half of the original state space), as well as those where they represent the same trace. Moreover, by comparing configurations (2) and (3), we separately identify the costs of both exploration phases. Compared to (2), (3) performs the second phase (identification procedure) in each cycle, while in (2), the second phase is nonexistent. For this configuration, the verification proves the *absence* of TAs (result: TRUE), which thus requires an exhaustive exploration of the possible variation in the first phase. We deduce from this comparison that the cost of the first phase is negligible wrt. that of the second phase (in this case, it represents less than 3 % of the total verification runtime). This confirms the need of implementing heuristics (such as $\texttt{H}_{\texttt{check}}$) for the crucial aspect of calling the identification procedure.

Table 8.1: Results of the detection of counter-intuitive TAs from the execution of TACLe [105] benchmarks on the OoO-pipeline formal model. We use a particular heuristic (H_{var}) to define the possible variations, based on the input program (cf. Sec. 8.3.1). "cex" stands for *counterexample* (result); °: cex showing a composition of variations, possibly moreover with no COM event among those at which a TA is triggered (°°). * stands for heuristics `asym'` or H'_{check} (cf. Sec. 8.3.1) instead of `asym` or H_{check}.

#	benchmark	N_{instr}	$\lvert mayDMiss \rvert$	*superscal*	`asym`	H_{check}	invariant	result	time	diam.	states found
1	countneg	50	4	4			true	true	00:00:14	59	4208
2					✓			true	00:00:09	59	2015
3							$noTA$	true	00:06:25	59	2105
4						*		true	00:00:36	59	2105
5						✓		true	00:01:24	59	2105
6				2				simple cex	00:00:12	43	728
7	iir	100	4	4	✓			cex°°	00:04:07	69	847
8						*		simple cex	00:03:58	100	4818
9						✓		simple cex	00:03:10	100	4749
10	cosf	30	7	4	✓	✓		true	00:23:24	86	118880
11				2				true	00:22:50	86	118991
12	fft	100	3	4	✓	✓		true	00:00:26	98	834
13	fir2dim	100	4	4	✓	✓		simple cex	00:05:42	81	1800
14	insertsort	30	6	4	✓	✓		cex°	00:00:31	29	3333
15					*			true	00:00:15	71	1159
16				2	✓			cex°°	00:00:26	30	3632
17					*			simple cex	00:00:17	37	456
18	complexup	100	4	4	✓	✓		simple cex	00:02:17	79	2563
19	bitonic	100	30	4	✓	✓		simple cex	00:05:21	35	18008

Detection of TAs

Configuration (4) implements $\texttt{H'}_{\texttt{check}}$, thus with a single call to our identification procedure (at the end of both executions), and brings a speedup in verification runtime of more than 90 % wrt. (3). With the main heuristic $\texttt{H}_{\texttt{check}}$ instead of $\texttt{H'}_{\texttt{check}}$, configuration (5) also leads to a speedup wrt. (3), but of less than 80 %. Indeed, the model checker performs fewer calls to the identification procedure, but more than one final call. However, the same reasoning on the second benchmark shows that the situation differs when there *actually exist* TAs: $\texttt{H'}_{\texttt{check}}$ (8) and $\texttt{H}_{\texttt{check}}$ (9) allow a speedup in verification runtime wrt. configuration (7), but the speedup in (9) is of about 23 % against less than 4 % in (8), thus considerably higher with heuristic $\texttt{H}_{\texttt{check}}$. Moreover, while the speedup offered by $\texttt{H}_{\texttt{check}}$ still represents 89 % of that offered by $\texttt{H'}_{\texttt{check}}$ in our case where there is *no* TA—configurations (4) and (5)—, the speedup offered by $\texttt{H}_{\texttt{check}}$ is 5.75 as important as that offered by $\texttt{H'}_{\texttt{check}}$ in our case where there *is* a TA—configurations (8) and (9). Consequently, heuristic $\texttt{H}_{\texttt{check}}$ allows a considerable speedup, whether there actually are TAs or not; this is consistent with the trade-off described in Sec. 8.3.1 and justifies that we only retain it for most examples.

Note that the cex for (7) shows complex execution scenarios ($^{\circ\circ}$), with more than a single favorable variation and thus, with an issue about composition, and without TAs triggered at COM events. We could wonder whether simpler scenarios could be found, by using `asym'` (to avoid complex compositions) or by relying on invariant $noTA(\text{TRUE})$ (to check for COM events at which TAs should be triggered). However, both cex derived from $\texttt{H}_{\texttt{check}}$ (8) and $\texttt{H'}_{\texttt{check}}$ (9) prove to be such simpler scenarios. They are simple counterexamples, i.e., similar to the traditional TA pattern, with a single favorable variation (no composition of variations) and at least one TA triggered at a COM event. Both simple counterexamples are different, even with the same diameter in the present case (which is sensible, since the property under consideration is not the same).

Let us now compare configurations (3)/(4)/(5) with (6). Through our parametric formal model, we can vary the microarchitectural parameters and base the verdicts of the detection on the concrete values. We observe that, although there is *no* counter-intuitive TA for $superscal = 4$, the execution of the first benchmark on a microarchitecture with $superscal = 2$ does entail TAs.

Some results from other benchmarks are reported in configurations (10), (11), and (12)—showing the *absence* of TAs—, and in (13), (18), and (19)—illustrating the *presence* of TAs through simple execution scenarios. In the case of configurations (10) and (11), contrary to that of (5) and (6), the verdict is the same for both values of parameter $superscal$. Fig. 8.10 represents a small portion of the generated graphs for both traces that constitute the cex of configuration (13).

Configuration (14) shows that, for the benchmark under question, a cex can be derived quickly. Although this cex clearly identifies TAs, we may intend to check for simpler scenarios, since this one is not easy to interpret ($^{\circ}$). The use of `asym'`

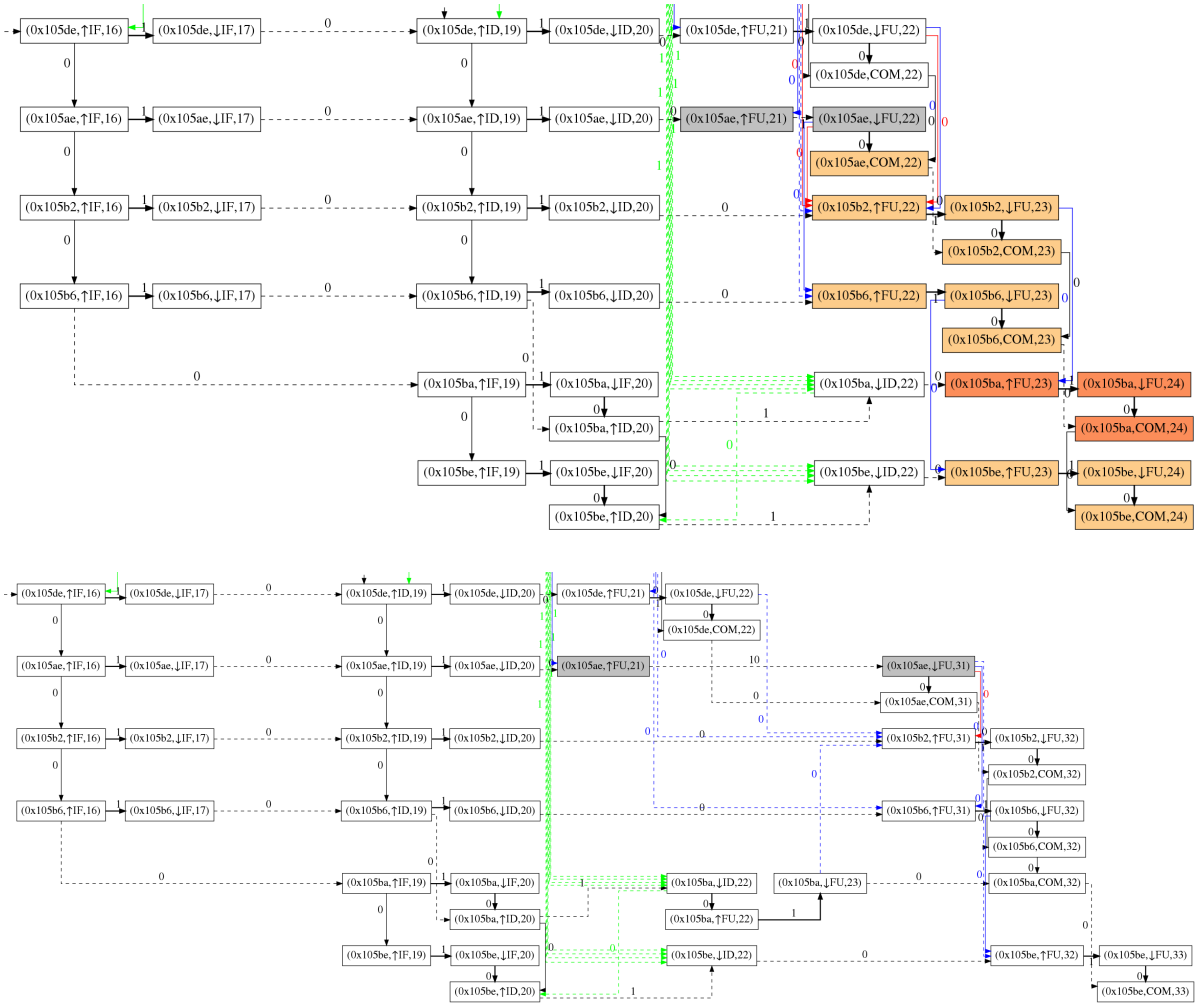


Figure 8.10: Portion of the graphs (ETDG/CG) produced during the automatic detection of counter-intuitive TAs for configuration (13) in Table 8.1, showing in particular the variation ( ) and the TAs that are triggered ( ).

in configuration (15) shows that, for this configuration, there do *not* exist simple scenarios (without cumulative effects of multiple variations) that trigger TAs.

A complex scenario is also found for the slightly different configuration (16), in which $superscal = 2$ instead of 4. Using `asym'` (17) leads to a simple scenario, with a single favorable variation and TAs triggered at COM events.

8.4 . Summary: a Practical Tool Support for the Accurate Detection of TAs

In this chapter, we proposed a detection procedure based on the formal definition of counter-intuitive TAs that we have introduced in the previous chapter. We thus

detailed the adaptations of the formal framework for implementing the procedure, and we presented the algorithm that constitutes the heart of the procedure.

We explained on short examples the key features of our procedure, in comparison with the application of the limited previous formal definitions from the literature, and we showed that these features match our intuitive understanding of TAs. We also highlighted that our framework allows for tackling the new, open problem of the composition of variations.

Finally, we exemplified on a standard benchmark collection how our procedure can serve to detect TAs on concrete applications. We presented the workflow that we set up so as to cover these benchmarks, and we reported some results for several configurations. **We have thus shown that our work provides a concrete *tool support* for the *accurate* detection of counter-intuitive TAs.**

Part IV
Heuristics for the Detection of Timing-Anomaly Patterns

In this last part, we tackle the second problem introduced at the end of the first part of the thesis, i.e., related to the formal modeling style, the abstractions, and the verification strategy that are suited to the detection of TA patterns in complex microarchitectures. The accurate identification of software-related patterns—thus taking into account the various microarchitectural features that impact the timing behavior—will allow for inserting counter-measures, in the purpose of mitigating the effects of TAs.

In this part, we focus on *amplification* TAs, but the same verification framework could be applied for the detection of counter-intuitive TAs. We made this choice since, to our knowledge, the state of the art explores the issue of deriving efficient abstractions for the detection of TAs only for amplification TAs [26]. Besides, although we believe that our formal framework proposed in the previous part and based on causality, for the detection of counter-intuitive TAs, might be adapted to capture amplification TAs, so far, only this work based on a typical delay scenario (cf. Sec. 2.1.2) offers a detection procedure for amplification TAs. We thus study amplification TAs starting from this work.

We investigate how the industrial superscalar TriCore microarchitecture (cf. Sec. 2.2) is amenable to compositional timing analyses, via a formal evaluation of the amplification TAs that can manifest in this microarchitecture. TriCore is a COTS (commercial off-the-shelf) hardware, in contrast to the simple microarchitectures on which amplification TAs were previously studied in the literature, and it actually suffers from such TAs. In this way, it becomes necessary to use formal verification to fully explore possible sources of amplifications.

Can amplification* TA*s be tracked efficiently for the TriCore microarchitecture? To answer this question, we develop in this part the following contributions:

1. **We adapt and extend the specialized abstraction called canonical pipeline model [26] (cf. Sec. 4.3.2), to capture the amplification effects in a formal model of the TriCore microarchitecture (Ch. 9) [4, 5].**

2. **We use model checking to efficiently detect amplification TAs and we report the associated complexity results (Ch. 9) [4, 5].**

3. **We aim for better accuracy as we design and implement counter-example-based methods, so as to uncover patterns leading to such anomalies (Ch. 10) [5].**

We model the TriCore microarchitecture using the language of UCLID5, we assess this model using the underlying BMC-based engine of UCLID5, and finally, we exploit the SMT back-end (Z3 solver) in order to get multiple counterexamples (cf. Ch. 3). Our formal model of TriCore and our script implementing these

heuristics are available on a GitHub repository.[10]

We show that detecting these anomalies over a model that fully represents possible dependencies between the TriCore pipelines is possible—however, only with appropriate reductions. We also provide a *collection of SMT-based heuristics*, for identifying the execution patterns that are likely to entail such anomalies. These heuristics properly address scalability issues in the detection of amplification TAs in TriCore.

CONTENTS

9 Detection of Amplification TAs . . . 165
9.1 Scale-up Modeling Process . . . 165
9.1.1 Adaptations for the Dual Pipeline of TriCore . . . 165
9.1.2 Progression and Stalling Logic . . . 168
9.1.3 Store Buffer . . . 172
9.1.4 WAW Hazards . . . 176
9.2 Evaluation of the TriCore Model . . . 177
9.2.1 Validation of the Model . . . 177
9.2.2 Results of the Detection . . . 178
9.3 Summary: our Modeling and Verification Approach . . . 184
10 Towards Software-Related Patterns . . . 185
10.1 Exploration of Multiple Counterexamples . . . 185
10.1.1 Delay Scenarios . . . 185
10.1.2 Specific SMT Problem . . . 186
10.2 Counterexample-Guided Exploration Strategies . . . 188
10.2.1 Broad-Spectrum State-Space Exploration . . . 190
10.2.2 Delay-Scenario Enumeration . . . 192
10.3 Evaluation . . . 193
10.3.1 Analysis of the Broad-Spectrum Exploration . . . 194
10.3.2 Analysis of the Delay-Scenario Enumeration . . . 198
10.4 Summary: a Step towards Accurate TA Patterns . . . 199

[10] https://github.com/t-crest/patmos-sail/tree/master/uclid/tricore

9 – DETECTION OF AMPLIFICATION TIMING ANOMALIES

Is the verification strategy from the canonical model (cf. Sec. 4.3.2) suitable for reliably detecting amplification TAs on TriCore? In this chapter, we present how to encode the abstraction of the canonical model for the TriCore microarchitecture and we provide a complete formalization of this microarchitecture. We rely on bounded model checking (cf. Ch. 3) to detect amplification TAs, based on a necessary condition. Due to the complexity of this microarchitecture, reductions are needed. These reductions efficiently remove useless pipeline configurations, wrt. timing anomalies, from the state space that has to be explored by the model checker. We analyze complexity results for different evaluation settings, with both generic and TriCore-specific reductions.

First, we provide, as a scale-up process from the canonical model, the full formalization of this specialized abstraction for amplification TAs, adapted for the more complex microarchitecture of TriCore (Sec. 9.1). This full formalization is the first step towards the obtainment of software-related TA patterns (cf. Ch. 10). Then, we address various TA-driven scenarios and we report the evaluation settings and results (Sec. 9.2).

9.1 . Scale-up Modeling Process

In this section, we introduce successive extensions of the canonical pipeline model (cf. Sec. 4.3.2). These extensions are not straightforward, as the canonical pipeline model requires both structural modifications (due to a second pipeline, Sec. 9.1.1), as well as functional extensions to accommodate the particularities of the TriCore microarchitecture wrt. the progression logic (Sec. 9.1.2), the store buffer (Sec. 9.1.3), and data dependencies (Sec. 9.1.4). As a whole, this section is the complete presentation of our formal modeling of TriCore.

9.1.1 . Adaptations for the Dual Pipeline of TriCore

In the canonical model, each instruction is characterized at a given instant by a tuple [26]:

$$\langle class, bl, stage, latency, delay, stalled, progress \rangle$$

where $class$ denotes the instruction class, bl the vector of *baseline latencies* for each pipeline stage, $stage$ the current pipeline stage, $latency$ the remaining amount of cycles to complete the baseline latency for the current pipeline stage, and $delay$ the accumulated delay in cycles, due to stalling so far. The first two attributes determine an *execution scenario* and *keep the same value after the initial state*. The last two attributes are Booleans related to the progression logic, where $stalled$

indicates that the instruction suffers a pipeline stall caused by *another* instruction, whereas $progress$ indicates whether the instruction may proceed to the next stage. Fig. 9.1 illustrates an instruction progressing through the pipeline.

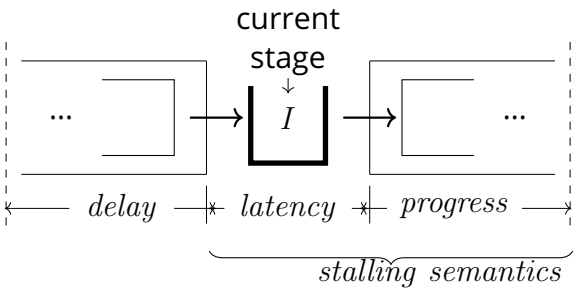


Figure 9.1: Canonical pipeline model (from [26]): the advancement of (upstream or downstream) instruction I in the pipeline is allowed by the $progress$ attribute, which depends on the $latency$ in its current stage and on the stalling semantics. The $delay$ attribute enumerates the number of cycles when I was stalled so far because of the other (downstream or upstream) instruction.

Since BMC is used, one must provide the bound up to which the model checker performs the transitions from one state to another (cf. Sec. 3.2.3). For the purpose of studying the effects of local variations for the downstream instruction, the model checker should perform transitions, i.e., advance the instructions through the pipeline, up to the completion of the downstream instruction in *all* behaviors. It is not necessary to check whether the upstream instruction could overtake the downstream instruction, nor delaying it, when the downstream instruction has finished its execution. The completion of the downstream instruction means that this instruction has reached the $post$ stage, which is specified by an LTL (see Sec. 3.1) property [26]:

$$\mathbf{G}(step = depth \implies stage_{down} = post) \tag{9.1}$$

where the $step$ state variable models time by counting the number of transitions (see Sec. 3.3.1), the current stage of the downstream instruction is $stage_{down}$ and the *specified* bound for BMC is $depth$. The LTL property ensures that the specified depth is high enough to reach all states where the current stage of the downstream instruction is $post$ (in any execution scenario). The *minimal* depth for which this property is verified is progressively established in a binary-search fashion, starting from arbitrary high values.

Jan et al. [26] state that a sufficient condition for the absence of amplification TAs is that the upstream instruction should never be able to delay the downstream instruction (e.g., contrary to the example of Table 2.2 that shows a TA). The violation of this property indicates that the total execution time of a sequence may not be compositional, i.e., *some hardware reasons prevent from considering instructions*

independently, and thus the combination of their timing effects. Whenever a downstream instruction is stalled, its $delay$ variable is incremented. If a strictly positive value of the downstream $delay$ is found by the model checker, there thus can be an amplification TA. The authors formulate the following LTL property, to which we further refer as the *delay property*. It states that a downstream instruction may never be delayed [26]:

$$\mathbf{G}(delay_{down} = 0) \tag{9.2}$$

It is a sufficient condition, as strictly positive delay values do not imply amplification TAs (i.e., there can be false positives) [26]. The generated counterexample, in particular the initial values of both the upstream and the downstream instructions, can be analyzed.

Addressing the dual pipeline of TriCore requires adapting both the model itself and the associated verification procedure.

Adaptation of the Canonical Model

The canonical pipeline model is designed for a single pipeline; therefore, its extension to accommodate the dual execution pipeline of TriCore requires some changes. As a direct consequence, the *upstream* and *downstream* instructions need to be *duplicated*, in order to capture the interactions between the I- and LS-pipelines of TriCore. **We thus consider four instructions**, denoted by $dw.p$ and $up.p$, where $p \in \{I, LS\}$. We also note that the dw and up instructions, respectively, do not necessarily belong to the same fetch bundle. Thus, in order to accurately model this TriCore-specific extension, we associate, to each instruction, three additional Boolean variables: $pbus$, $dbus$, and $conflict$. Their semantics is as follows: $pbus$ and $dbus$ indicate, respectively, whether the instruction needs to access the SRI bus through the PMI and/or DMI, and $conflict$ captures if the instruction is currently subjected to interference due to priority rules (which are to be detailed below). Thus, $pbus$ and $dbus$ are set whenever the required instruction/data do not reside in the *local* cache or scratchpad memory. These two attributes, initialized with non-deterministic values (cf. Sec. 3.3.1), refer to specific stages and are constant, i.e., their values remain unchanged after the initial state. We also note that the $dbus$ attribute for an instruction that is neither a load nor a store (in particular an I-instruction) is not set. The $conflict$ attribute is part of the progression logic (it is computed and it is not constant).

Adaptation of the Verification Procedure

The verification procedure [26] is adapted likewise. The $up.LS$ instruction serves as the reference point and is initially placed in the pre stage, while the three other instructions can be freely placed in any pipeline stage. The only constraint is that the (older) instruction $dw.p$ has to be more advanced than the instruction $up.p$, $p \in \{I, LS\}$, in the respective pipeline, according to the total order of stages $\mathcal{S}$ defined in Sec. 9.1.2. As a consequence, we obtain the following two properties,

adapted from formulae (9.1) and (9.2):

$$\mathbf{G}(step = depth \implies (stage_{dw.LS} = post \wedge stage_{dw.I} = post)) \tag{9.3}$$

$$\mathbf{G}(delay_{dw.LS} = 0 \wedge delay_{dw.I} = 0) \tag{9.4}$$

Property (9.3) extends Property (9.1) as the *downstream* instructions of both pipelines should terminate their execution, whereas Property (9.4) expresses a sufficient condition for the absence of amplification TAs in the TriCore microarchitecture, as it is originally stated for a single pipeline, by Property (9.2).

9.1.2 . Progression and Stalling Logic

In the canonical model [26], when an instruction enters a pipeline stage, its $latency$ is initialized with the corresponding value in bl. Then, every transition (specified in the `next` block) after which the instruction has to remain at least one more cycle in the stage decrements the $latency$ of the instruction—unless the progression logic stalls the instruction due to another instruction in the pipeline. The baseline latencies (bl) thus are only *minimal* latencies, as some conditions may require stalling and thus delay instructions more than they may require themselves (as illustrated in Table 2.2). The result ($progress$) from the progression logic is based both on the remaining cycles ($latency$) and on the conditions that may induce stalling ($stalled$). In addition to the actual pipeline stages of the processor, the models contain two special stages, pre and $post$, where instructions reside before entering the actual pipeline and after completion, respectively. Both pipeline stages have initial latencies of 1, as instructions can enter the pipeline as soon as possible and $post$ is always the last possible stage.

We must now specify the progression logic for TriCore. The structure of both pipelines (see Fig. 2.2) is modeled as a totally ordered set of stages:

$$\mathcal{S} = \{\ pre, \mathrm{IF}, \mathrm{ID}, \mathrm{EX}, \mathrm{EX2}, \mathrm{WB}, \mathrm{SB}, post\ \}$$

where stages $\mathrm{EX2}$ and SB (for store buffer) are optional for the I- and LS-pipelines, respectively—as shown in Fig. 2.2.

Progression Logic for Upstream Instructions

Formula (9.5) specifies under which conditions an $up.p$ instruction ($p \in \{I, LS\}$) is allowed to advance to the next stage in the TriCore model:

$$progress'_{up.p} \triangleq latency_{up.p} \leq 1 \wedge \neg stalled'_{up.p} \wedge \neg conflict_{up.p} \wedge next_{up.p} \neq stage'_{dw.p} \tag{9.5}$$

This formula is expressed using the standard notation for state transitions (see Ch. 3), where primed variables (e.g., $progress'_{up.p}$) indicate next state values, while unprimed variables refer to current state values. We use the sign $\triangleq$ for variable assignments and the sign $=$ for variable comparisons.[1] Variable $latency$ (cf.

[1] As mentioned in Sec. 3.3.1, for better readability, we present here our specification in the form of first-order formulae.

$$common_{dw} \triangleq (\exists p \in \{I, LS\}\colon stage_{dw.p} = IF \wedge pbus_{dw.p} \wedge latency_{dw.p} > 1) \vee (stage_{dw.LS} = EX \wedge dbus_{dw.LS} \wedge latency_{dw.LS} > 1) \quad (9.6)$$

$$stalled'_{up.LS} \triangleq common_{dw} \vee (stage_{up.I} = IF \wedge pbus_{up.I} \wedge latency_{up.I} > 1) \quad (9.7)$$

$$stalled'_{up.I} \triangleq common_{dw} \vee (stage_{up.LS} = IF \wedge pbus_{up.LS} \wedge latency_{up.LS} > 1) \vee (stage_{up.LS} = EX \wedge dbus_{up.LS} \wedge latency_{up.LS} > 1) \quad (9.8)$$

$$conflict_{up.LS} \triangleq (stage_{up.LS} = pre \wedge pbus_{up.LS}) \wedge (stage_{dw.LS} = ID \wedge dbus_{dw.LS}) \quad (9.9)$$

$$conflict_{up.I} \triangleq (stage_{up.I} = pre \wedge pbus_{up.I}) \wedge [(\exists x \in \{dw, up\}\colon stage_{x.LS} = ID \wedge dbus_{x.LS}) \vee (stage_{dw.LS} = pre \wedge pbus_{dw.LS})] \quad (9.10)$$

Figure 9.2: Formulae capturing the particular progression logic for *upstream* instructions.

Sec. 4.3.2) is used to check whether the instruction has completed its execution in the current stage (e.g., completed its *own* cache miss or bus access). The Boolean variable $stalled$ indicates stalling due to the progression logic, which may prevent the $up.p$ instruction from advancing, e.g., due to another instruction experiencing a cache miss. The newly introduced $conflict$ attribute is used to model the interference on the SRI bus. Finally, the $up.p$ instruction may only advance if the corresponding *downstream* instruction, $dw.p$, does not occupy the $next$ stage. We also note that $next$ represents the next pipeline stage of the instruction, not always the stage at the next step.

Formula (9.5) defines the **progression logic** of the dual pipeline, whose specificities are captured by the attribute $stalled$. Formulae (9.6) to (9.10), which capture the details of the progression logic for upstream instructions, are grouped together under Fig. 9.2 in order to increase readability. Formulae (9.6) to (9.8) illustrate for the *upstream* instructions one variant of such a progression logic, called *whole* [26] (cf. Sec. 4.3.2), where both pipelines are stalled simultaneously whenever one of the pipelines needs to stall. Consequently, the $stalled$ variables characterize the stalling situations due to *in-progress* accesses, which occur whenever a bus access has *already* begun and is not about to complete. The stalling of each pipeline is expressed through a common stalling condition given by Formula (9.6). As such, $common_{dw}$ denotes stalling of the *upstream* instructions in the I- and LS-pipelines due to the *downstream* instructions. The instructions involved in the stalling must access the SRI bus (the $pbus$ or $dbus$ attributes are

set). It is important to note that only load operations may lead to baseline latencies that are higher than 1. Due to this property, the $class$ attribute is not tested in association with the current stage being EX.

The $up.LS$ instruction may only experience additional stalling when $up.I$ is in the IF stage (the second disjunct in Formula (9.7)), while the $up.I$ instruction may experience such stalling when $up.LS$ is in the IF stage (the second disjoint in Formula (9.8)) or in the EX stage (the second line of Formula (9.8)). We address the stalling due to store instructions in the next section, where we present the modeling of the store buffer.

Next, we address **the interference of the shared SRI bus**. The $conflict$ variables, e.g., in Formulae (9.9) and (9.10), capture the stalling due to interference, in particular due to *upcoming* accesses. The SRI bus can be configured with different arbitration policies, including round-robin and fixed-priority. We model a fixed-priority scheme for the SRI bus, where data memory accesses are prioritized over instruction fetches.

Formula (9.9) indicates that the (*upstream*) LS instruction may only experience interference on the SRI bus during instruction fetch, thus being stalled in the pre stage, which is expressed by the first conjunct of (9.9). This moreover may only occur when the *downstream* instruction $dw.LS$, in the ID stage, is to start its execution (in the EX stage) and to access the SRI bus, expressed by the second conjunct of (9.9). The $up.I$ instruction may also experience interference during instruction fetch from the $dw.LS$ and $up.LS$ instructions when either of them is about to enter the EX stage while accessing the SRI bus, expressed by the first disjunct in (9.10). The second disjunct addresses the particular case where the $up.I$ and $dw.LS$ instructions are in the same fetch bundle, but span over two cache lines or scratchpad accesses.

Pipeline Progression

An (upstream or downstream) $x.p$ instruction advancement (where $x \in \{dw, up\}$, $p \in \{I, LS\}$) in the pipeline is captured by Formula (9.11), according to the value of the related $progress'_{x.p}$:

$$progress'_{x.p} \implies (stage'_{x.p} \triangleq next_{x.p} \wedge latency'_{x.p} \triangleq bl_{x.p}.next_{x.p}) \tag{9.11}$$

Whenever an instruction can advance one step in the pipeline, the $stage_{x.p}$ attribute is updated with its new value $next_{x.p}$, and the $latency_{x.p}$ attribute (for the remaining cycles in the current stage) is initialized with the baseline latency $bl_{x.p}$ of the stage. Otherwise, the $stage_{x.p}$ attribute remains unchanged, as specified by Formulae (9.12) and (9.13).

When an instruction cannot advance to the next stage and is not experiencing stalling by another instruction, it must advance its execution *in the current stage*, while decrementing its (remaining) latency in the stage, as in Formula (9.12):

$$[\neg progress'_{x.p} \wedge \neg stalled'_{x.p}] \implies (stage'_{x.p} \triangleq stage_{x.p} \wedge latency'_{x.p} \triangleq latency_{x.p} - 1) \tag{9.12}$$

$$common_{up} \triangleq (\exists p \in \{I, LS\}\colon stage_{up.p} = IF \wedge pbus_{up.p} \wedge latency_{up.p} > 1) \\ \vee (stage_{up.LS} = EX \wedge dbus_{up.LS} \wedge latency_{up.LS} > 1) \quad (9.15)$$

$$stalled'_{dw.LS} \triangleq common_{up} \vee (stage_{dw.I} = IF \wedge pbus_{dw.I} \wedge latency_{dw.I} > 1) \quad (9.16)$$

$$stalled'_{dw.I} \triangleq common_{up} \vee (stage_{dw.LS} = IF \wedge pbus_{dw.LS} \wedge latency_{dw.LS} > 1) \\ \vee (stage_{dw.LS} = EX \wedge dbus_{dw.LS} \wedge latency_{dw.LS} > 1) \quad (9.17)$$

$$conflict_{dw.LS} \triangleq \bot \quad (9.18)$$

$$conflict_{dw.I} \triangleq (stage_{dw.I} = pre \wedge pbus_{dw.I}) \\ \wedge [(\exists x \in \{dw, up\}\colon stage_{x.LS} = ID \wedge dbus_{x.LS}) \\ \vee (\exists x \in \{dw, up\}\colon stage_{x.LS} = pre \wedge pbus_{x.LS})] \quad (9.19)$$

Figure 9.3: Formulae capturing the particular progression logic for *downstream* instructions.

Finally, whenever an instruction cannot progress since it is stalled by another instruction, both $stage_{x.p}$ and $latency_{x.p}$ remain unchanged, as in Formula (9.13):

$$[\neg progress'_{x.p} \wedge stalled'_{x.p}] \implies (stage'_{x.p} \triangleq stage_{x.p} \wedge latency'_{x.p} \triangleq latency_{x.p}) \quad (9.13)$$

Progression Logic for Downstream Instructions

The progress formula for the *downstream* instructions, in (9.14), is slightly simpler than its counterpart for the *upstream* instructions, given in (9.5). Explicit checking whether the next stage is available is unnecessary, due to the ordering of the instructions.

$$progress'_{dw.p} \triangleq \ latency_{dw.p} \leq 1 \ \wedge \ \neg stalled'_{dw.p} \wedge \ \neg conflict_{dw.p} \quad (9.14)$$

Similar stalling conditions stand for the *downstream* instructions. The details about the progression logic for downstream instructions are given by Formulae (9.15) to (9.19), which are grouped together under Fig. 9.3. Formulae (9.16) and (9.17) are expressed based on a common stalling condition, in (9.15), whenever SRI bus accesses occur. Similar to the stalling of the *upstream* instructions, expressed by Formulae (9.7) and (9.8), the second disjunct in (9.16) and the last two disjuncts in (9.17) represent the stalling of the *downstream* instructions by the *downstream* instruction in the other pipeline, due to PMI accesses in Formula (9.16), and either PMI or DMI accesses in Formula (9.17).

The priority rules for the *downstream* instructions are complementary with those of the *upstream* instructions. Formulae (9.9) and (9.10) express that the

$dw.LS$ instruction has a higher priority wrt. the *upstream* instructions, regardless of PMI or DMI accesses. Consequently, interference scenarios involving an *upstream* instruction are not expressed in Formula (9.18), corresponding to the conflicts experienced by the (*downstream*) LS instruction. As for the $dw.I$ instruction, it does not have priority, even in the case of simultaneous PMI accesses, as previously expressed by Formula (9.9). Consequently, the formula of $conflict_{dw.I}$ must explicitly contain all the possible interference scenarios with the $up.LS$ instruction.[2] Formula (9.18) states that the $conflict$ attribute of the $dw.LS$ instruction is always false. In particular, the priority rule about data memory accesses and instruction fetches prevents $dw.LS$ from experiencing interference on the SRI bus from the $up.LS$ instruction. This is consistent with the *upstream* Formula (9.9).

Formula (9.19) captures all the possible SRI bus interference between the $dw.I$ instruction and both $up.LS$ and $dw.LS$ instructions. Such interference occurs only when $dw.I$ performs accesses through the PMI, expressed by the first conjunct of (9.19). Moreover, data memory accesses are prioritized and may come from either the $up.LS$ or the $dw.LS$ instruction, as expressed by the first disjunct of (9.19). The particular case of instructions from the same fetch bundle entailing delays are addressed with Formulae (9.10) and (9.19) (i.e., the second disjunct).

The cycles leading to the stalling of a *downstream* instruction due to an *upstream* instruction are accumulated into the $delay$ attribute:

$$delay'_{dw.p} \triangleq delay_{dw.p} + [\neg progress'_{dw.p} \wedge stage'_{dw.p} \neq post \wedge (stalled'_{dw.p} \vee latency'_{dw.p} = 0)] \tag{9.20}$$

Formula (9.20) states that the $delay_{dw.p}$ variable of a *downstream* instruction is incremented if the instruction cannot progress (first conjunct in the second line), provided that this particular instruction is still in the pipeline (i.e., not in the $post$ stage, cf. the second conjunct), either because it is stalled by another instruction (i.e., the $stalled_{dw.p}$ variable) or because of a conflict (third conjunct). In this latter case, the remaining latency of the instruction has elapsed. Note that we have adopted here the convention where *true* terms evaluate to 1 and *false* to 0.

9.1.3 . Store Buffer

We model the asynchronously operating store buffer as a special pipeline stage, SB, placed at the end of the LS-pipeline between the WB and $post$ stages, as shown in Fig. 2.2. Asynchronous memory accesses of SB are modeled using the regular baseline latencies associated with the SB stage, which are similar to those of load instructions (in the EX stage).

[2] It cannot interfere with the $up.I$ instruction, since both instructions may only access the PMI in the IF stage.

Fill Status

The store buffer of the TriCore microarchitecture contains several entries. Multiple store instructions may thus place their data into the store buffer without stalling—as long as the store buffer has free entries. Consequently, and in contrast to regular stages, the $up.LS$ and $dw.LS$ instructions can *both* simultaneously occupy entries in the store buffer. While we do not model the actual fill level (number of entries occupied), it suffices to introduce a new Boolean instruction attribute *prio_SB* that is associated with $up.LS$ and $dw.LS$ and may be valid for store instructions. This variable is set whenever a store instruction performs a memory access and the SB is full. It is a constant attribute, initialized with a non-deterministic value.

The model checker consequently explores all possible scenarios where the buffer's fill status impacts the arbitration between SRI bus accesses. A store may only stall the pipeline (for its writing operation) when the buffer is full (i.e., *prio_SB* is set).[3] Whenever the *prio_SB* attribute of a store instruction is not set, this instruction may stall in the WB stage.

SRI Bus Conflicts

The store buffer may asynchronously access the SRI bus, potentially causing additional bus conflicts. It has the lowest priority and only starts new transfers when the bus is idle (i.e., after stalling in the WB stage). An ongoing transfer from the store buffer may cause interference when the bus was initially idle. In addition, the store buffer has the highest priority whenever it is full. These new forms of bus conflicts need to be added, as disjuncts, in formulae (9.9)-(9.10) and (9.18)-(9.19). Moreover, the existing conflicts related to instructions in the ID stage only apply to loads; for example, an expression such as $(stage_{dw.LS} = ID \wedge dbus_{dw.LS})$ must be replaced by: $(stage_{dw.LS} = ID \wedge class_{dw.LS} = load \wedge dbus_{dw.LS})$.

New conflicts arise from store instructions that are currently in the WB stage and about to enter SB, when the $dbus$ and *prio_SB* (i.e., the store buffer *is* full) attributes are set. These conditions indeed prevent store instructions from stalling in the WB stage in this case. The formulae that capture these conflicts are refined accordingly with all these considerations and are presented under Fig. 9.4.

Upstream instructions. Formulae (9.21) and (9.22) present the refined $conflict$ conditions for the *upstream* instructions. The $up.I$ instruction accessing the bus through the PMI could experience conflicts from an ongoing transfer of the store buffer (through the DMI) (third disjunct in (9.21)) or from a forthcoming transfer (when the store buffer is full) (last disjunct in (9.21)). Thus, the $up.I$ instruction experiences conflicts before entering the IF stage (in pre).

The $up.LS$ instruction accessing the PMI may experience the same conflicts (due to the $dw.LS$ instruction). Moreover, as shown in Formula (9.22), a forthcoming ($up.LS$) load may experience exactly the same conflicts. Finally, we also

[3]This will be refined in Sec. 9.1.3 by taking into account the case of dependent loads (Formula (9.26)).

$$
\begin{aligned}
conflict_{up.I} \triangleq\ & (stage_{up.I} = pre \wedge pbus_{up.I}) \\
& \wedge [(\exists x \in \{dw, up\}\colon stage_{x.LS} = ID \wedge class_{x.LS} = load \wedge dbus_{x.LS}) \\
& \quad \vee (stage_{dw.LS} = pre \wedge pbus_{dw.LS}) \\
& \quad \vee (\exists x \in \{dw, up\}\colon stage_{x.LS} = SB \wedge dbus_{x.LS} \wedge latency_{x.LS} > 1) \\
& \quad \vee (\exists x \in \{dw, up\}\colon stage_{x.LS} = WB \wedge class_{x.LS} = store \wedge dbus_{x.LS} \wedge prio\_SB_{x.LS})]
\end{aligned} \tag{9.21}
$$

$$
\begin{aligned}
conflict_{up.LS} \triangleq\ & \{(stage_{up.LS} = pre \wedge pbus_{up.LS}) \wedge [(stage_{dw.LS} = ID \wedge class_{dw.LS} = load \wedge dbus_{dw.LS}) \\
& \qquad \vee (stage_{dw.LS} = SB \wedge dbus_{dw.LS} \wedge latency_{dw.LS} > 1) \\
& \qquad \vee (stage_{dw.LS} = WB \wedge class_{dw.LS} = store \wedge dbus_{dw.LS} \wedge prio\_SB_{dw.LS})]\} \\
& \vee \{(stage_{up.LS} = ID \wedge class_{up.LS} = load \wedge dbus_{up.LS}) \wedge [(stage_{dw.LS} = SB \wedge dbus_{dw.LS} \wedge latency_{dw.LS} > 1) \\
& \qquad \vee (stage_{dw.LS} = WB \wedge class_{dw.LS} = store \wedge dbus_{dw.LS} \wedge prio\_SB_{dw.LS})]\} \\
& \vee \{(stage_{up.LS} = WB \wedge class_{up.LS} = store \wedge dbus_{up.LS} \wedge \neg prio\_SB_{up.LS}) \\
& \qquad \wedge [\exists x \in \{dw, up\}\colon stage_{x.I} = pre \wedge pbus_{x.I}]\}
\end{aligned} \tag{9.22}
$$

$$
\begin{aligned}
conflict_{dw.I} \triangleq\ & (stage_{dw.I} = pre \wedge pbus_{dw.I}) \\
& \wedge [(\exists x \in \{dw, up\}\colon stage_{x.LS} = ID \wedge class_{x.LS} = load \wedge dbus_{x.LS}) \vee (\exists x \in \{dw, up\}\colon stage_{x.LS} = pre \wedge pbus_{x.LS}) \\
& \quad \vee (\exists x \in \{dw, up\}\colon stage_{x.LS} = SB \wedge dbus_{x.LS} \wedge latency_{x.up} > 1) \\
& \quad \vee (\exists x \in \{dw, up\}\colon stage_{x.LS} = WB \wedge class_{x.LS} = store \wedge dbus_{x.LS} \wedge prio\_SB_{x.LS})]
\end{aligned} \tag{9.23}
$$

$$
\begin{aligned}
conflict_{dw.LS} \triangleq\ & (stage_{dw.LS} = WB \wedge class_{dw.LS} = store \wedge dbus_{dw.LS} \wedge \neg prio\_SB_{dw.LS}) \\
& \wedge [(\exists x \in \{dw, up\}\colon stage_{x.I} = pre \wedge pbus_{x.I}) \vee (stage_{up.LS} = pre \wedge pbus_{up.LS}) \\
& \quad \vee (stage_{up.LS} = ID \wedge class_{up.LS} = load \wedge dbus_{up.LS})]
\end{aligned} \tag{9.24}
$$

Figure 9.4: *Refined* formulae capturing the priority rules of interference due to the *store buffer*.

need to consider that an ($up.LS$) store instruction may experience conflicts in the WB stage, when the store buffer is *not* full. In this case, only the I-pipeline may cause interference (through the PMI), since the $dw.LS$ instruction is more advanced in the pipeline.

Downstream instructions. A similar reasoning applies for the *downstream* instructions. Formula (9.23), for example, refines the conflict terms for the *downstream* instructions, as expressed by (9.19). The additional terms are the same as in the $conflict_{up.I}$ formula. A $dw.LS$ store instruction may only experience conflicts related to the SB in the WB stage, when the store buffer is *not* full. Moreover, the $dw.LS$ instruction cannot interfere with a store transfer of $up.LS$, because it is more advanced in the pipeline than the $up.LS$ instruction. This single new form of conflict must be added as a disjunction with $\perp$ in Formula (9.18), which actually leads to the refined Formula (9.24). This formula states that a $dw.LS$ store instruction in the WB stage, expressed by the first conjunct, may suffer interference from fetching both *upstream* instructions, from a *downstream* instruction $dw.I$ or from a memory load by an *upstream* instruction $up.LS$.

Store Buffer and Dependent Loads

In the pipeline model, a hazard caused by a dependent load may occur when $dw.LS$ is a store instruction in the WB or SB stage, and $up.LS$ is a load instruction in the EX stage (see Sec. 2.2.2).[4] Our pipeline model needs to distinguish whether both instructions refer to the same address, namely this is the case of a memory-reference hazard. We address this particular point using a new global Boolean attribute $memdep$ to indicate, when valid, that $dw.LS$ and $up.LS$ are two dependent load/store instructions. The model checker again explores all possible assignments of this variable—while respecting consistency, i.e., the two instructions are of the required (load and store) class. Moreover, as the data cache has a write-allocate policy [32], the dependent load always experiences a cache hit, i.e., $dbus_{up.LS}$ is not set:

$$memdep \implies \neg dbus_{up.LS} \tag{9.25}$$

Since the dependent load stalls the pipeline, the access of the store buffer to the SRI bus is prioritized—similar to the case when the store buffer is full. Consequently, the $prio\_SB_{dw.LS}$ attribute is set:

$$(class_{dw.LS} = store \wedge class_{up.LS} = load) \implies prio\_SB_{dw.LS} \tag{9.26}$$

Finally, we model the impact of these stalls by extending the progression logic as follows:

$$\ldots \vee (stage_{up.LS} = EX \wedge memdep \wedge stage'_{dw.LS} \neq post)$$

[4] Note that for the specific handling of hazards (cf. Sec. 2.2.2), the progression logic differs from the *whole* logic.

Formula (9.5) is refined into (9.27) such that, in the case of the $up.LS$ instruction, the new stalling condition does not refer to *another* instruction but to $up.LS$ itself:

$$\begin{aligned} progress'_{up.LS} \triangleq\ & latency_{up.LS} \leq 1 \wedge \neg stalled'_{up.LS} \\ & \wedge \neg conflict_{up.LS} \wedge next_{up.LS} \neq stage'_{dw.LS} \\ & \wedge \neg(stage_{up.LS} = EX \ \wedge memdep \ \wedge stage'_{dw.LS} \neq post) \end{aligned} \tag{9.27}$$

The additional expression prevents an *upstream* store instruction to progress through the pipeline until the *downstream* has completed its write transfer, i.e., is progressing to the $post$ stage. Note that the $memdep$ attribute cannot be set if the instruction classes are not $load$ and $store$; hence, the instruction classes are not tested in the previous expression:

$$(class_{dw.LS} \neq store \vee class_{up.LS} \neq load) \implies \neg memdep \tag{9.28}$$

In addition, we update the current latency of the EX stage on a specific condition that is given by Formula (9.29). The dependent load instruction only performs the memory access *after* the other memory access, of the store instruction, terminates. Consequently, the latency is decremented once the store access is completed. Formula (9.12) is thus refined with a specific version for the $up.LS$ instruction that captures the advancement from any stage of this instruction. This formula is obtained by substituting in Formula (9.12) the antecedent (between square braces $[\cdot]$) with the condition of Formula (9.29):

$$\begin{aligned} & [\neg progress'_{up.LS} \wedge \neg stalled'_{up.LS} \wedge \\ & \neg(stage_{up.LS} = EX \wedge memdep \wedge stage'_{dw.LS} \neq post)] \end{aligned} \tag{9.29}$$

The specific $up.LS$ version of Formula (9.13) (when the latency remains unchanged) is refined in accordance by substituting the antecedent with Formula (9.30):

$$\begin{aligned} & [\neg progress'_{up.LS} \wedge \neg stalled'_{up.LS} \wedge \\ & (stage_{up.LS} = EX \wedge memdep \wedge stage'_{dw.LS} \neq post)] \end{aligned} \tag{9.30}$$

9.1.4 . WAW Hazards

Write-after-Write (WAW) dependencies entail interactions between *both* pipelines; more precisely they may delay the progression in the LS-pipeline (cf. Sec. 2.2.2). Similarly to dependent loads, a set of new attributes is associated with the instructions of the LS-pipeline ($waw\_dw_{x.LS}$ and $waw\_up_{x.LS}$, $x \in \{up, dw\}$), indicating a WAW dependency from an $upstream$ load or a $downstream$ load to either $dw.I$ or $up.I$, respectively. The progression logic of instructions in the LS-pipeline is then extended, as presented in Formulae (9.31) and (9.32) in Fig. 9.5, to encode the various scenarios by adding a disjunct to Formulae (9.7) and (9.16) for $up.LS$ and $dw.LS$.

$$stalled'_{up.LS} \triangleq common_{dw} \vee (stage_{up.I} = IF \wedge pbus_{up.I} \wedge latency_{up.I} > 1)$$
$$\vee (stage_{up.LS} = ID \wedge [(waw\_dw_{up.LS} \wedge (stage_{dw.I} = ID \vee stage_{dw.I} = EX))$$
$$\vee (waw\_up_{up.LS} \wedge (stage_{up.I} = ID \vee stage_{up.I} = EX))]) \quad (9.31)$$
$$stalled'_{dw.LS} \triangleq common_{up} \vee (stage_{dw.I} = IF \wedge pbus_{dw.I} \wedge latency_{dw.I} > 1)$$
$$\vee (stage_{dw.LS} = ID \wedge [(waw\_dw_{dw.LS} \wedge (stage_{dw.I} = ID \vee stage_{dw.I} = EX))$$
$$\vee (waw\_up_{dw.LS} \wedge (stage_{up.I} = ID \vee stage_{up.I} = EX))]) \quad (9.32)$$

Figure 9.5: *Refined* formulae capturing the progression logic of (upstream and downstream) instructions, due to WAW *hazards*.

Formulae (9.31) and (9.32) state that the $up.LS$ and $dw.LS$ instructions in the ID stage stall when they depend either on the $dw.I$ or the $up.I$ instructions. The total number of stall cycles is iteratively determined by the current stage (ID or EX) of the related instruction, which may progress meanwhile.

9.2 . Evaluation of the TriCore Model

We conducted two types of experiments on our model of the TriCore microarchitecture. The first benefit drawn from these experiments is the validation of the model (Sec. 9.2.1). The second type of experiments concerns the evaluation of amplification TAs in our TriCore model (Sec. 9.2.2).

9.2.1 . Validation of the Model

In the first type of experiments, we *empirically* validate the specification presented in Sec. 9.1. We are interested in the cycle-accurate timing behavior of the microarchitecture. To the best of our knowledge, available simulation tools for TriCore, such as TSIM (TriCore Instruction Set Simulator) [107, p.45] or $\mathrm{TRACE32}$ Instruction Set Simulator [108], are driven by the functional specification, thus at the instruction level. As far as we are concerned, we rely on documentation, in particular the description of the timing behavior of the TriCore pipelines [33]. We test our model in order to check that it is compliant with the pipeline description [33] in all the documented scenarios. Note that we do not aim at being exhaustive in our modeling, since we focus on the effects arising from the shared bus (for instance, we do not model all instruction opcodes). We produce traces on our model from trivial invariants, entailing a trace output and we check that the output is conform to the documentation. Such experiments allow us to establish confidence in our model during its development.

For instance, we fixed the instruction classes so that $class_{dw.LS} \triangleq store$ and

$class_{up.LS} \triangleq load$ (with arbitrarily fixed values for the two other instructions) and we observed the traces describing an unrolled execution up to a certain depth, both with $memdep$ not set and set. We checked that when this variable is set, the behavior of our model specified by Formulae (9.27) and (9.29) is conform to the timing behavior described in Table 2.4b, extracted from the documentation [33, p.7]. Similarly, for instance, we fixed $class_{up.LS} \triangleq load$ and $class_{up.I} \triangleq mac$, as well as the initial stage for $up.I$ ($up.LS$ being initialized at pre): $stage_{up.I} \triangleq pre$ (same fetch bundle) or $stage_{up.I} \triangleq ID$ (i.e., previous fetch bundle). In each case, we produced traces with both $waw\_up_{up.LS}$ set or not set. We thus checked here that the part of the behavior specified by Formula (9.32) is conform to the description made in Tables 2.5b and 2.5d, from the documentation [33, pp.6, 14].

We adopted this approach for all the cases covered by the model. Moreover, we also checked that all counterexamples found by the further assessments are consistent, e.g., the execution scenarios reported in Table 9.2 (Sec. 9.2.2) and Fig. 10.2 (Sec. 10.3). An example of a refinement of the model in the wake of the analysis of such a counterexample is detailed in Sec. 10.3.2.

9.2.2 . Results of the Detection

In the following experiments, we will assess amplification TAs in (variants of) the TriCore microarchitecture, using the verification strategy from Sec. 9.1.1. **Aside from expanding the results of the existing comparative study [26], with a more complex microarchitecture, our work also evaluates the scalability of the canonical model for tracking amplification TAs.** Then, we evaluate possible refinements where data dependencies are considered, we analyze a counterexample showing an amplification TA and finally, we address a code-specific extension. Table 9.1 reports the runtime of the model checker (column *Runtime*) for various microarchitecture models (column *Core model*). It mentions various configurations of the TriCore model (TRx) and of the basic in-order pipeline [26] for comparison (INx). The model configurations differ in terms of the modeled core features and reduction strategies that are applied (cf. the check marks ✓). For these experiments, the bounded model checker explores a minimal number of steps (see Sec. 3.3.1) for the downstream instructions to reach the $post$ stage (see Sec. 9.1.1), as indicated by column *Min. bound*.

Evaluated Core Models

In order to proceed with an incremental evaluation of the scalability of the model, we firstly evaluate the TriCore model without data dependencies.

TriCore Adaptation. Without data dependencies, instructions within the isolated I-pipeline do not interfere with the LS-pipeline. Consequently, the only stage in the I-pipeline that may affect the execution is IF. Since stalling entails the same stall cycles in the LS-pipeline, modeling instruction progressions in the I-pipeline is not relevant wrt. TAs. Their initial $stage$ is thus set to $post$ in the

Table 9.1: Evaluation of various configurations (i.e., modeled features and reduction strategies) of the TriCore model (TRx), compared to the basic in-order model with the *whole* progression logic [26] (INx).

		Features			Reductions					
Core model	Configuration	Store buffer	Mem. dep.	WAW	General	Interference	WAW	Code-spec.	Min. bound	Runtime (h:min:s)
Basic In-order	(IN1)								33	0:00:29
	(IN2)				✓				33	0:00:15
TriCore (single pipeline)	(TR1)	✓				✓			38	0:06:42
	(TR2)	✓			✓	✓			38	0:05:14
	(TR3)	✓	✓		✓	✓			38	0:05:23
TriCore (dual pipeline)	(TR4)	✓	✓		✓				53	8:23:04
	(TR5)	✓	✓	✓	✓	✓			>53	>7:23:53
	(TR6)	✓	✓	✓	✓	✓	✓		53	6:52:15
	(TR7)		✓	✓	✓	✓			53	3:48:35
	(TR8)		✓	✓	✓	✓	✓		53	3:38:38
	(TR9)	✓	✓	✓	✓	✓	✓	✓	42	0:28:49

simplest configurations of the TriCore model. This feature is referred to as *Interference reduction* in Table 9.1. The resulting model configurations are quite similar to the single-pipeline models [26]; hence, they are qualified as *single-pipeline* configurations. Table 9.1 provides the result for the whole-logic basic in-order pipeline (IN1) [26] (with the same verification support as for TriCore), and for the basic TriCore model (TR1). The significant difference in runtimes—from seconds (IN1) to minutes (TR1)—is due to the state space of the TriCore model being extended by the multiple possible baseline latencies for the additional stage SB representing the store buffer (see Sec. 9.1.3).

General Reductions. The previous case studies [26] were simpler than the TriCore model and hence, aggressive state space reductions were not required. This point changes in the case of the TriCore model, as the verification of this particular model requires efficient verification strategies. The verification procedure must explore all possible baseline latencies for each stage. First, varying the values of baseline latencies (bl) of stages that will never be reached by an instruction, because its initial stage is greater in the pipeline order, is useless for the verification procedure. These values are thus systematically enforced to one at the initial step:

$$\forall x \in \{up, dw\},\ \forall p \in \{I, LS\},\ \forall s \in \mathcal{S},\ (s < stage_{x.p} \implies bl_{x.p}.s = 1) \quad (9.33)$$

This kind of reductions is implemented in $\mathrm{UCLID5}$ as *assumptions* (see Sec. 3.3.1). For instance, if the initial stage is different than pre or IF, the latency of IF is 1.

Furthermore, all values in the *range* defined by the maximal value of the possible baseline latencies of the *initial* stage must be preserved. For instance, a complete 10-cycle latency for a memory access actually entails the exploration of all the values ($bl_{dw.LS}.EX$) that are ranged between 1 and 10, if the (downstream LS) instruction is initialized in the EX stage. Indeed, the instruction may have already completed partially its latency for its current stage (EX) before the $up.LS$ instruction is about to enter the pipeline (pre in the initial state). Nevertheless, the state space could be reduced by retaining only *concrete* baseline latencies for non-initial stages. For instance, an instruction in the IF stage should use, in the EX stage, the particular baseline latencies of Table 2.3b, instead of all the range defined by the maximal value (10).

Configurations (TR2) and (IN2) in Table 9.1 restart (TR1) and, respectively, (IN1) with added *General reductions*. Though the absolute differences are not substantial, the relative (20 % and 48 %) gains on execution times are significant. The preserved partial baseline latencies for SB are responsible for the lower global decline as to the TriCore model.

Data Dependencies

The specific refinements of the TriCore model are evaluated hereafter.

Memory References. Configuration (TR3) shows that the additional verification time when adding data memory references to the TriCore model (by relaxing

the value of the $memdep$ attribute now possibly set) is only of 9 seconds. Actually, as explained in Sec. 9.1.3, the single situation allowing this kind of dependencies is a load following a store, with the baseline latency for the load in EX excluding cache misses by design.

Dual Pipeline. Dealing with WAW dependencies (as in the next paragraph) requires the second pipeline in the model, marked in Table 9.1 by *dual pipeline*. Configuration (TR4) shows a state space explosion and a significantly higher runtime, even in the absence of WAW dependencies. Though modeling a dual pipeline (thus including the I-pipeline) without WAWs does not impact amplification anomalies, this configuration exposes the associated state space explosion. On the one hand, this setting generates multiple cases with different baseline latencies for fetching in the I-pipeline, and on the other hand, it causes more stalling cases of both pipelines.

WAW Dependencies. Some reductions are thus necessary before modeling WAW dependencies. Note that the I-pipeline may interfere with the LS-pipeline when actual WAWs are explored. In that case, the following Boolean attributes may be set and the *Interference reduction* needs to be refined:

$$\begin{aligned}(\neg waw\_up_{up.LS} \wedge \neg waw\_up_{dw.LS}) &\iff stage_{up.I} = post \\ (\neg waw\_dw_{up.LS} \wedge \neg waw\_dw_{dw.LS}) &\iff stage_{dw.I} = post\end{aligned} \tag{9.34}$$

The reduction holds only when an instruction in the I-pipeline does not interfere with the LS-pipeline, namely it does not cause dependencies for any LS-instructions. However, the previous depth of 53 is not enough to verify the property. Indeed, hazards cause more stall cycles and more transitions to complete. (TR5) already takes more than 7 hours, despite the *Interference reduction*. Besides, WAW hazards can only happen in ID and EX. It is not necessary to explore the (no-effect) occurrences of data dependencies if these stages are out of reach. At the initial step, we thus assume the so-called WAW *reduction*:

$$\begin{aligned}stage_{dw.LS} > ID &\implies (\neg waw\_up_{dw.LS} \wedge \neg waw\_dw_{dw.LS}) \\ stage_{up.I} > EX &\implies (\neg waw\_up_{up.LS} \wedge \neg waw\_up_{dw.LS}) \\ stage_{dw.I} > EX &\implies (\neg waw\_dw_{up.LS} \wedge \neg waw\_dw_{dw.LS})\end{aligned} \tag{9.35}$$

The first implication means that the downstream LS-instruction is considered independent of both upstream and downstream I-instructions if it is initialized after the ID stage. The last two implications mean that I-instructions after the EX stage do not cause dependencies for any LS-instructions. These implications have a side effect when combined with Formulae (9.2.2), since they transitively boil down to replacing all the targeted stages by $post$, i.e., applying the *Interference reduction* in an efficient way. With the WAW *reduction* (TR6), the verification with possible WAW hazards still requires the former bound of 53 and a quite sensible verification runtime.

Assessment of the Store Buffer. We notice that, when comparing configurations (TR1) and (IN1), the presence of the SB in configuration (TR1) entails a considerable overhead, in comparison to the basic in-order pipeline (IN1) with an otherwise comparable feature set. This is further emphasized by configuration (TR7), which re-confirms the role of the SB model on the state space size. When stores are excluded from the possible values of instruction classes to be explored (through an assumption in the initial state), the verification with possible WAW hazards (TR5) does not require a greater bound, nor some specific reductions. This verification process completes even faster than the full exploration with the WAW *reduction* of configuration (TR6). Furthermore, configuration (TR8) shows that the WAW *reduction* mainly affects stores. Indeed, the verification times with the reduction (TR8) or without the reduction (TR7) are similar in the *absence* of the SB. This shows that the main side effect of the WAW *reduction* on the *Interference reduction* is due to the states involving a store instruction in the SB. The numerous possible baseline latencies associated with this stage entail a significant state space explosion.

Analysis of a Counterexample

Table 9.2 presents a counterexample returned by the verification procedure on configuration (TR6). This example is based on two multiply-accumulate instructions, $up.I$ and $dw.I$, and a load following a store ($up.LS$ and $dw.LS$), with the store buffer not full. The load does not depend on the store but has a WAW dependency with $up.I$. The data writing in SB ($dw.LS$) and the data reading in EX ($up.LS$) have each a baseline latency of 5 cycles, representing scratchpad accesses through the SRI bus (see Table 2.3b). The pipeline progression is conform to the above specification, in particular with the $dw.LS$ instruction stalled in the WB stage due to the *whole* progression logic, which does not apply to the WAW dependency.

The presence of a TA is confirmed when another execution scenario with the same instruction classes, same order, same dependencies, and same initial stages, but without delays, can be deduced from the counterexample. Table 9.2 shows such a scenario (*), whose behavior in the I-pipeline is identical but in which the load instruction does not need to access the SRI bus, since data are accessible after a cache hit. Due to the store buffer and its conflict rule, this scenario entails a global timing variation $\Delta_G = t_{14} - t_9 = 5$ larger than the local variation $\Delta_L = 5 - 1 = 4$ relative to the data operation of the load (in Table 2.3b).

Note that similar counterexamples without data dependencies can be generated by the model checker, under the specified assumptions. Unlike the SB, data dependencies do not introduce new types of TAs. The delays introduced by the related stalls are not the primary sources of TAs, since they are not due to *unknown* hardware states while executing the code. Our model accurately represents the occurrence of data dependencies and the timing behavior of the counterexample from Table 9.2.

Table 9.2: A counterexample provided by the verification procedure on TriCore (TR6) and a deduced counterpart (*) where the (LS) downstream instruction is not delayed by the upstream instruction, confirming an amplification TA.

pipeline	instr.	*class*	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
I	*dw*	*mac*	ID	EX	EX2	WB	*post*										
	up	*mac*	IF	ID	EX	EX2	WB										
LS	*dw*	*store*	IF	ID	EX	WB	WB	WB	WB	WB	WB	SB	SB	SB	SB	SB	*post*
	up	*load*	*pre*	IF	ID	ID	EX	EX	EX	EX	EX	WB	*post*	*post*	*post*	*post*	*post*
LS *	*dw*	*store*	IF	ID	EX	WB	SB	SB	SB	SB	SB	*post*					
	up	*load*	*pre*	IF	ID	ID	EX	WB	*post*	*post*	*post*	*post*					

Code-Specific Verification

This approach can be specialized to be *code-specific*, by restricting the non-determinism for the successive instructions (class, latencies...). Such software reductions tend to represent the execution of one particular input program. On the other hand, a code-specific approach requires modeling the ISA more finely. In such a model, even though the hardware is known to exhibit amplification TAs, the execution of a given program might hide them.

The aforementioned verification configurations generate *generic* counterexamples. When we take into consideration a model of the application code, we should expect more *specific* counterexamples and even to prove the absence of amplification TAs wrt. a path of this code. A first step towards generating specific counterexamples requires fixing the types of the four instructions and their code order (through $\mathrm{UCLID5}$ assumptions). Configuration (TR9), for example, restarts configuration (TR6) after restricting both I-instructions to be of class multiply-accumulate (with loads as LS-instructions). The total order imposed on the four instructions is the following:

$$stage_{up.LS} \leq stage_{up.I} \leq stage_{dw.LS} \leq stage_{dw.I}$$

which is consistent with the previously described partial order. As expected, the set of counterexamples does not include the store buffer and the case of Table 9.2. Also, the execution time is more than 14 times less than in configuration (TR6), advocating for a code-specific TA detection. Recall, however, that this detection deals with one program path only.

9.3 . Summary: our Modeling and Verification Approach

We proposed a formal and executable model of TriCore, a sophisticated microarchitecture that we introduced in Ch. 2 and that is used in the automotive industry. Our model is specialized to evaluate real-time systems wrt. amplification TAs. We extended the existing abstraction called canonical pipeline model (cf. Ch. 4), by considering both structural and functional elements of the TriCore microarchitecture: stalling logic, store buffer, data dependencies. We specified each of these elements and we evaluated them with the formal verification framework of $\mathrm{UCLID5}$ (cf. Ch. 3).

We showed how to achieve a scalable detection of amplification TAs by integrating appropriate reductions in the TriCore model: we must take advantage of the hardware specificities to remove useless configurations from the state space. We also showed that a code-specific detection is valuable in order to reduce the verification runtime.

10 – TOWARDS THE IDENTIFICATION OF SOFTWARE-RELATED PATTERNS

In this chapter, we aim at exploring the various execution scenarios that lead to downstream instructions being delayed (i.e., delay scenarios). Our approach advances the systematic study of amplification TAs in two directions. First, it enables us to identify code sequences that do not exhibit TAs when executed on the TriCore microarchitecture. Second, it analyses the potential sources of TAs. In future work, this should help integrate the undesired timing behavior in WCET analysis and deploy counter-measures to limit their occurrence.

We thus monitor and guide the verification engine towards covering the state space in desired ways, aiming to obtain multiple counterexamples related to the delay property of Formula (9.4). Our abstraction (the adapted canonical pipeline model) is too coarse to determine accurate execution sequences, as it relies on a limited number of representative instructions. We propose refinement strategies towards constructing software-related execution patterns showing TAs, and their projection on our pipeline model of TriCore.

First, we formulate the problem that consists in getting multiple counterexamples (Sec. 10.1). Then, we detail the implementation of several SMT-based strategies (Sec. 10.2), and finally, we present the assessment of those strategies on the TriCore model (Sec. 10.3).

10.1 . Exploration of Multiple Counterexamples

Hereafter, we explain why we need to adapt the verification procedure in order to obtain multiple counterexamples (Sec. 10.1.1) and how we handle the generated SMT problems in this pursuit (Sec. 10.1.2).

10.1.1 . Delay Scenarios

The *delay scenario* of a counterexample generated from Formula (9.4) is the state corresponding to the smallest depth at which a downstream instruction exhibits a strictly positive delay in the counterexample. The anatomy of such a counterexample shows a *prefix*, i.e., a series of states leading to the delay scenario, the delay scenario itself, and a *suffix*, where either the delay value is incremented again or the instructions advance through the pipeline (or stay in the $post$ stage).

There exist counterexamples from one transition after the initial state, i.e., delay scenarios found at a depth of one. Consequently, one run of the BMC engine *up to* a certain bound basically provides a set of counterexamples of different

lengths (see Sec. 3.3.1). However, these counterexamples might actually share the same prefixes and delay scenarios. In this case, the execution in the pipeline is more or less unrolled. Moreover, different scenarios might exist at the same depth, from different initial states, and thus remain unexplored. **We must guide the verification in order to derive new counterexamples and delay scenarios (whatever the depth)**, as detailed in Sec. 10.2.

The *whole* progression logic (see Sec. 9.1.2), though realistic for the TriCore model, entails many stall cycles, shared by several instructions, and leads to long counterexamples. Moreover, these stall cycles are likely to give false positives from Property 9.2 (cf. Sec. 9.1.1), which would require a thorough inspection of the counterexamples. In order to better exemplify the proposed strategies for the multiple counterexample generation, we implement the so-called *only-upstream* logic [26], adapted for the TriCore model. With this progression logic, upstream instructions are still stalled whenever an instruction performs an SRI bus access, while downstream instructions can advance through the pipeline if an upstream instruction performs a bus access. Note that a downstream instruction may still suffer interference from the SRI bus when the instruction itself requires the bus. We also refine the transition relation for the $delay$ variables, initially in Formula (9.20), so as to restrict the delay scenarios to cases where a downstream instruction cannot progress through the pipeline though it has finished its execution in the current stage. We thus remove the $stalled$ attribute in the disjunction of Formula (9.20): the delays are incremented only when the $latency$ has elapsed (or is about to elapse).

10.1.2 . Specific SMT Problem

Hereafter, we implement specialized methods to control the provided counterexamples. Instead of letting the model checker call the SMT solver, we export the SMT problems into files and then manipulate them through the Z3 API (see Sec. 3.3.1). We dispose of functions **sat**, **model** and **assert**, which allow, respectively, deciding whether the problem is satisfiable, getting an SMT model,[1] i.e., an interpretation m (cf. Def. 3.8) that satisfies the problem, and adding new SMT assertions as logical formulae (see Table 10.1).

The advantages of this direct manipulation of counterexamples/SMT models are twofold. First, it simplifies the interaction with the SMT solver, when adding new assertions based on the derived counterexamples. We furthermore dispose of the **push** and **pop** functions (cf. Sec. 3.3.1), which allow us to conveniently add and remove assertions during the exploration (see Table 10.1). Second, we can target *one* specific depth, in particular the bound specified in $\mathrm{UCLID5}$, and avoid iterating up to this particular depth. We can work on the last iteration of the unrolled model, thus a *single* SMT file corresponding to the bound. Note,

[1] In this chapter, we call 'SMT model' a model *of a formula* (cf. Def. 3.9) and we reserve the phrase "model" for the formal specification (of TriCore).

Table 10.1: Notations used in the counterexample-guided exploration strategies (Algorithms 10.1 to 10.3).

Variables and SMT problem	
$\mathcal{X}$	Set of state variables defined in the pipeline specification (Ch. 9).
$\mathcal{I} \subset \mathcal{X}$	Set of *initial conditions* (constant state variables, see Sec. 10.1).
$\mathcal{IS} = \mathcal{I} \cup \{stage_{dw}, stage_{up.I}\}$	Set of *extended initial conditions* (with $stage$ variables, see Sec. 10.1).
max_d	Bound of the BMC problem described by the input problem smt_pb.
$\mathcal{X}_d$	Set of SMT variables representing a valuation of the state variables at depth d.
$\mathcal{V} = \bigcup_{d \in \{0,\dots,max\_d\}} \mathcal{X}_d$	Set of SMT variables representing the state variables over all depths.
$x_d \equiv x$	$x_d \in \mathcal{X}_d$ is a depth-level SMT variable representing state variable $x \in \mathcal{X}$.
$\mathcal{F}$	Set of all first-order formulae with $\mathcal{V}$ as symbols of variables.
$\mathcal{P}$	Designates the power set.
Special functions *implicitly operating* on the input variable $smt\_pb \in \mathcal{P}(\mathcal{F})$	
sat : $\mathcal{P}(\mathcal{F}) \to \{\top, \bot\}$	$\top$ if the SMT problem is satisfiable, $\bot$ otherwise (relies on the solver).
model : $\mathcal{P}(\mathcal{F}) \to (\mathcal{V} \to \mathcal{D})$	Get an (SMT) model of the SMT problem from the solver.
Operators and functions on SMT variables/models	
$\mathcal{Y}[d \in \mathbb{N}] : \mathcal{Y} \cap \mathcal{X}_d$	Subset of the terms in $\mathcal{Y}$ that represent a valuation of state variables at depth d.
$\mathcal{Y}\{x \in \mathcal{X}\} : \bigcup\{x_d \in \mathcal{Y} \mid x_d \equiv x\}$	Subset of the terms in $\mathcal{Y}$ that represent (depth-level) values of state variable x.
$[\![\cdot]\!]_m : \mathcal{V} \to \mathcal{D}$	Interpretation of an SMT variable in model m (i.e., a suitable value in domain $\mathcal{D}$).
$delay\_depth : (\mathcal{V} \to \mathcal{D}) \to \mathbb{N}$	Minimal depth s.t. the $delay_{dw}$ variable is not null: $\min\{d \in \mathbb{N} \mid \mathcal{V}[d]\{delay_{dw}\} \neq (0,0)\}$.
Special functions *implicitly updating* the input variable smt_pb **(**$\mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{F})$**)**	
assert	Update the SMT problem with a new assertion.
push	Add next assertions into the stack (new scope) (Algorithm 10.2).
pop	Remove previous assertions from the stack (scope end) (Algorithm 10.2).
Other notations (Algorithm 10.2)	
list	Get an (arbitrarily ordered) list from a set.
rem	Remove the first element of a list.
yield	Return a value from a (Python) generator and set the reentry point at the next statement.

however, that this SMT file contains the unrolled terms related to the valuations of the state variables, at *each* depth up to the bound. The symbols in $\mathcal{V}$ represent the variables of the SMT formula, i.e., all the valuations of the state variables in $\mathcal{X}$ (see Sec. 3.2.3). The operators denoted by $\mathcal{Y}[d \in \mathbb{N}]$ and $\mathcal{Y}\{x \in \mathcal{X}\}$ allow the extraction of all the terms related to depth d (whatever the state variable) and, respectively, all the terms related to the values of state variable $x \in \mathcal{X}$ (whatever the depth) (see Table 10.1). A delay scenario derived from the file may thus actually occur at *any* depth of the unrolled model. Function $delay\_depth$ allows finding this depth from a given SMT model (see Table 10.1).

The set of all state variables is denoted by $\mathcal{X}$ (see Table 10.1). Some state variables of the TriCore model remain constant after their initialization, for example those corresponding to instruction attributes (for any instruction when appropriate[2]): $class$, bl (Sec. 4.3.2), $pbus$, $dbus$, $prio\_SB$, $memdep$, waw_dw and waw_up (Sec. 9.1.1). These state variables form the subset $\mathcal{I}$ of *initial conditions* (see Table 10.1). These variables, as well as those of the $stage$ attribute (which may be updated on each transition), are not explicitly initialized and hence can assume any non-deterministic values in their domains, except for $stage_{up.LS}$, initialized with a fixed value. All these state variables, which form the *extended initial conditions* $\mathcal{IS}$ (see Table 10.1), introduce non-determinism through the assumptions governing the possible values in the *initial state* (see Sec. 3.3.1). Consequently, one valuation of all these variables ($\mathcal{IS}$) in the initial state totally determines an unrolled execution up to a given bound. The value of a variable $x \in \mathcal{IS}$ in the initial state, in particular, can thus be accessed through: $\mathcal{V}[0]\{x\}$. The other state variables, e.g., of the attributes $latency$, $delay$, $stalled$, and $progress$, are computed according to relations between other state variables and may be updated on each transition.

We propose different methods to explore the state space and to identify patterns that exhibit TAs when executed on the TriCore model. While these methods differ in the way that the explored counterexamples are evaluated, they all share a first step, that of fixing the bound, i.e., the maximal depth—and thus *one* SMT file. This depth must be sufficiently large to include *all* the counterexamples and guarantees that no interaction between the upstream and the downstream instructions is omitted, i.e., it must verify the property of Formula (9.3). We take the lowest depth verifying this property (i.e., similar to the minimal bounds reported in Table 9.1).

10.2 . Counterexample-Guided Exploration Strategies

Since non-determinism is expressed only in the initial states (Sec. 10.1.2), we can focus on the SMT symbols related to the extended initial conditions, in order to exclude the SMT models that assign all of them the same values as previously

[2]For instance, the $memdep$ attribute globally characterizes two instructions.

explored. Those symbols are later referred to as *initial terms*. This method is described by Algorithm 10.1, which shows the blueprint for the development of two other exploration methods (described below and assessed in Sec. 10.3).

Algorithm 10.1: Basic counterexample-guided exploration strategy

Input: $smt\_pb, \mathcal{X}, \mathcal{I}, \mathcal{IS}, \mathcal{V}$

1 **while sat do**
2 $\quad m \leftarrow$ **model**;
3 $\quad$ **forall** $x \in \mathcal{IS}$ **do**
4 $\quad\quad$ **print** $[\![\mathcal{V}[0]\{x\}]\!]_m$;
5 $\quad d \leftarrow delay\_depth(m)$;
6 $\quad$ **forall** $x \in \mathcal{X} \setminus \mathcal{I}$ **do**
7 $\quad\quad$ **print** $[\![\mathcal{V}[d]\{x\}]\!]_m$;
8 $\quad$ **assert** $\neg \left(\bigwedge_{x \in \mathcal{IS}} \mathcal{V}[0]\{x\} = [\![\mathcal{V}[0]\{x\}]\!]_m\right)$;

Algorithm 10.1 takes as input the set of terms $\mathcal{V}$ in particular, thus notably the downstream delays for each depth.[3] The algorithm iterates until there are no more counterexamples (line 1), meaning that all initial conditions entailing a TA have been explored. While the problem at hand is still satisfiable, the algorithm gets (line 2) an SMT model and prints/saves the values of all the initial terms (lines 3 and 4). Algorithm 10.1 focuses on the extended initial conditions, with the initial terms being used to *block* the current initial state, i.e., evict it from the next possible initial states, through additional SMT assertions (line 8). We note that these values do not provide information about the delay scenario and consequently, Algorithm 10.1 aims to determine the minimal depth where the downstream delay has a positive value (line 5). Moreover, this algorithm prints/saves the values of terms at this depth where the delay scenario occurs (lines 6 and 7). It suffices to consider the non-constant terms, i.e., corresponding to the state variables in $\mathcal{X} \setminus \mathcal{I}$. Lines 2 to 7 are shared by the subsequent two algorithms that we designed in this context of counterexample-guided exploration for amplification TAs.

The drawback of Algorithm 10.1 lies in its poor runtime performance, expected since it depends on the number of counterexamples. There are many initial conditions to lead to the same delay scenario, e.g., due to different baseline latencies in the *same* initial stage: the complex memory hierarchy of TriCore implies that many baseline latencies exist for the same stage. However, Algorithm 10.1 serves as **a blueprint for other, more efficient, counterexample-driven exploration heuristics**. As expected from such heuristics, the sets of explored counterexamples are not complete but meet particular criteria, presented hereafter.

[3] This delay $delay\_dw$ is here actually a tuple (one delay per I or LS downstream instruction) and a positive delay value is a tuple different from the $null\_delay = (0, 0)$.

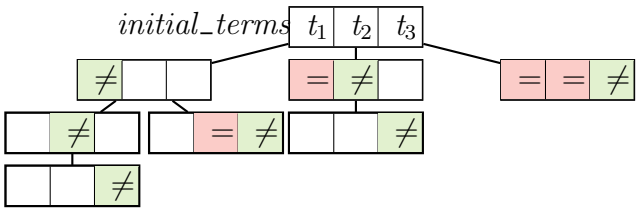


Figure 10.1: Principle of the broad-spectrum exploration on a simplified three-term example, where each edge represents a call to the SMT solver and each node represents the additional blocking ($\neq$) or fixing ($=$) constraints asserted on term values wrt. their parents.

10.2.1 . Broad-Spectrum State-Space Exploration

This first proposed heuristic is a refinement of Algorithm 10.1, aiming to detect delay scenarios from various extended initial conditions. In essence, we implement a procedure based on scopes and incremental solving (see Sec. 3.3.1) for approximating an exhaustive solver [66].

Principle. Fig. 10.1 maps the procedure on a simplified case study, made of $n = 3$ initial terms. Starting with a satisfiable set of assertions and a subsequent SMT model (i.e., the root of the tree), the state space is split into $n = 3$ domains that are guaranteed to be disjoint (i.e., first level in the tree). The i-th domain has the first $i - 1$ terms *blocked* from taking the same values (denoted by $\neq$) and the i-th term *fixed* with the same value (denoted by $=$) as in the root. The first node blocks the first term, so that it cannot take the same value as in the root, and thus makes the domain disjoint from the root. The second node fixes the value of the first term (with the same value as in the root) and blocks the value of the second term, making the domain disjoint from the root and the previous node. Finally, the last node fixes the first two terms (disjoint from the previous two nodes) and blocks the third term (disjoint from the root). These three nodes represent new assertions imposed on the solver: one blocking and several fixing assertions for each node. For each new domain, if the problem with the additional set of assertions remains satisfiable, a new SMT model is derived. The same process is started over in a new scope, taking into account *only* the $n - i$ terms without additional constraints so far. In Fig. 10.1, when deriving the second level in the tree, only the last two terms are considered from the first node of the first level, only the third term from the second node, whereas the last node is terminal. Switching between branches means defining new scopes through push/pop operations on assertions. Scopes are destroyed only on switching; hence the current assertions are kept from one node to its children (not represented in Fig. 10.1). In this way, *all* domains are guaranteed to be disjoint.

This procedure cannot distinguish more than two counterexamples that differ only in a *single* term. It is, however, a sufficient argument for the noticeable

improvement of the solving runtime.

Algorithm 10.2: Broad-spectrum incremental exploration

Input: $smt\_pb, \mathcal{X}, \mathcal{I}, \mathcal{IS}, \mathcal{V}$

```
1  Function broadExpl(list_of_terms)
2      if sat then
3          m ← model;
4          forall x ∈ IS do
5              print [[V[0]{x}]]_m;
6          d ← delay_depth(m);
7          forall x ∈ X \ I do
8              print [[V[d]{x}]]_m;
9          yield m;
10         forall term ∈ list_of_terms do
11             push;
12             assert term ≠ [[term]]_m;
13             forall t <_list_of_terms term do
                   /* <_l means that the rank in list l is smaller.   */
14                 assert t = [[t]]_m;
15             new_list ← rem(list_of_terms);
16             forall m ∈ broadExpl(new_list) do
17                 yield m;
18             pop;
19 init_list ← list(⋃_{x ∈ IS} V[0]{x});
20 forall m ∈ broadExpl(init_list) do
21     yield m;
```

Application to the Problem. Algorithm 10.2 describes the procedure exemplified in Fig. 10.1 with the specific terms of our SMT problem. It is based on a recursive function, which is called at the beginning of a new branch. Each call (except the one in line 20) is accordingly framed by a scoping push/pop pair. This function takes as input the list of remaining terms on which it should still operate. The first call (line 20) is made with the list of *all* initial terms (from extended initial conditions). The function checks whether there is at least one counterexample/SMT model. If so, an SMT model is obtained (line 3) and the initial terms (lines 4 and 5) and the delay scenario (lines 7 and 8) are printed/saved. The function terminates (line 9) with this first SMT model as its result.

Algorithm 10.2 was implemented in Python and the main function, *broadExpl*, was coded as a generator (i.e., a function with the behavior of an iterator). The

generator *yields* the current value, i.e., it returns the value but reenters at the next statement when it resumes (see Table 10.1).[4] Due to the loop iterating over the generator (line 20), the function re-enters at line 10. Here, the outer loop (lines 10 to 18) represents the origin of all the branches from the root (Fig. 10.1), i.e., all the content of each loop iteration is confined within a scope. Within these scopes, it adds the (first) new assertions of the branches (lines 12 to 14), namely the blocking assertion and the fixing assertions of the child in the current branch.

The recursive calls of this function (line 16) exhaustively explore all the children of the current branch; this function terminates when there are no more counterexamples in the current branch. Since a new iterator object is created on line 20, those calls make the function re-enter at the beginning (line 2) where the satisfiability of the problem at hand is checked. In the case that this problem is unsatisfiable, the exploration of the current branch is over and the current iterator reaches its end. Its caller can continue its execution, performing pop operations on the branch assertions (line 18). If the problem remains satisfiable, an SMT model is derived (line 3) and this process re-starts for the next children. Finally, each new iterator prepares all further additional assertions in the branch (loop on line 10) for it to consider one less term, as in Fig. 10.1.

10.2.2 . Delay-Scenario Enumeration

Though the previous method is designed to provide a broad spectrum of counterexamples in an efficient way, it is incomplete (in the sense that it cannot enumerate all the delay scenarios of the TriCore model). Moreover, since many extended initial conditions can lead to the same delay scenario, the procedure described by Algorithm 10.2 provides sets of counterexamples that share many similarities. These counterexamples slightly differ in terms of the initial conditions (e.g., baseline latencies) and may, in fact, have the same delay scenarios. The next heuristic extends the initial-state exploration of Algorithm 10.2 to a more accurate exploration of counterexamples, which is based on the enumeration of the delay scenarios.

Algorithm 10.3 is similar to Algorithm 10.1, as it iterates until the SMT model under consideration becomes unsatisfiable. The terms concerning the $delay_{dw}$, $stage_{dw}$, and $stage_{up}$ state variables play a particular role, since they may *characterize* a delay scenario. As previously presented in Algorithm 10.1, the delay scenario is extracted (lines 5 to 7), for each SMT model. This does not only print/save the counterexamples (by exporting the values, line 7), but also serves to guide the next solving iterations (as in Algorithm 10.1). In that respect, we do not fix any term, but *block* the delay scenario. Since the same delay scenario might happen again at any depth (see Sec. 10.1.1), we block it considering the relevant terms corresponding to *any depth*. We thus prevent further counterexamples from having, at any depth, the same combination of stages as in the current SMT model m, as soon as the associated delay has a positive value (line 10).

[4] https://docs.python.org/3/glossary.html#term-generator

Algorithm 10.3: Delay-Scenario Enumeration

Input: $smt\_pb, \mathcal{X}, \mathcal{I}, \mathcal{IS}, \mathcal{V}, max\_d$

while sat do
  $m \leftarrow$ **model**;
  forall $x \in \mathcal{IS}$ **do**
    print $[\![\mathcal{V}[0]\{x\}]\!]_m$;
  $d \leftarrow delay\_depth(m)$;
  forall $x \in \mathcal{X} \setminus \mathcal{I}$ **do**
    print $[\![\mathcal{V}[d]\{x\}]\!]_m$;
  forall $d' \in \{1, ..., max\_d\}$ **do**
    assert $\neg(\mathcal{V}[d']\{delay_{dw}\} \neq (null\_delay = (0,0))$
      $\wedge\, \mathcal{V}[d']\{stage_{dw}\} = [\![\mathcal{V}[d]\{stage_{dw}\}]\!]_m$
      $\wedge\, \mathcal{V}[d']\{stage_{up}\} = [\![\mathcal{V}[d]\{stage_{up}\}]\!]_m)$;

Contrary to Algorithm 10.2, Algorithm 10.3 does not incrementally construct its solution. Its efficiency comes from the fact that it handles less counterexamples than Algorithm 10.1 (i.e., due to the fact that there are fewer delay scenarios than initial conditions).

10.3 . Evaluation

Hereafter, we assess the broad-spectrum exploration (Algorithm 10.2) and the delay-scenario enumeration (Algorithm 10.3) on the TriCore model. We present the results derived from two core model configurations, described in Table 9.1: the code-specific one (TR9) and the more general one (TR6).[5]

Table 10.2 presents relevant statistics wrt. the state-space exploration of both aforementioned algorithms on both model configurations: the code-specific configuration (Table 10.2a) and the more general configuration (Table 10.2b). **The broad-spectrum exploration is faster in both configurations of the model. This method is particularly efficient in finding numerous counterexamples.** For this method, the number of calls to the solver (so as to check whether a certain set of assertions is satisfiable) is greater than the number of the provided counterexamples. This is indeed due to the fact that Algorithm 10.2 does not stop when a certain SMT problem becomes unsatisfiable, but continues its exploration on another branch, representing an unexplored part of the state space.

Table 10.3 enumerates all the counterexamples for the code-specific configuration, and Table 10.4, for the general configuration.

[5] With the *only-upstream* progression logic, however (see Sec. 10.1.1).

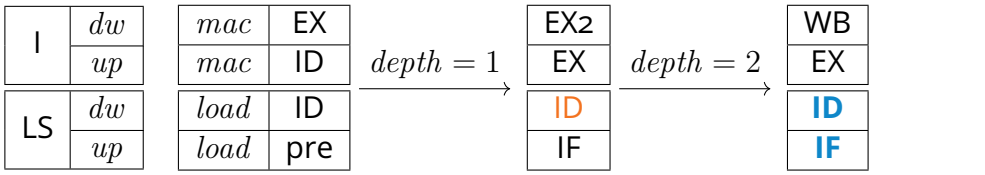
Table 10.2: Overview of the generation of multiple counterexamples (cex.).

(a) Code-specific configuration (features of core model TR9)

Method	Calls to Z3	Number of cex.	Total time (s)
Algorithm 10.2	134	64	331.5
Algorithm 10.3	13	13	379

(b) General configuration (features of core model TR6)

Method	Calls to Z3	Number of cex.	Total time (s)
Algorithm 10.2	626	218	2950.3
Algorithm 10.3	64	64	4721.1



(a) Initial conditions **(b)** Intermediate state **(c)** Delay scenario

Figure 10.2: Interpretation of the counterexamples derived from our procedures based on SMT solving.

10.3.1 . Analysis of the Broad-Spectrum Exploration

Presentation of the Results. Tables 10.3a and 10.4a present the results from the broad-spectrum method, in terms of initial conditions and delay scenarios (in the two columns). Move precisely, both tables show the valuation of the initial terms, making up extended initial conditions that lead to an amplification TA, and the valuation of relevant state variables at the depth at which the delay scenario occurs. Many actual counterexamples differ in their initial terms due to several factors: the bus accesses, the baseline latencies, and the WAW attributes. For brevity, we regroup the counterexamples under shared values of the *stage* terms, for the initial conditions and the delay scenario, all at once. Hence, each line represents as many counterexamples as described by the multiplicity column and these counterexamples share the same initial stages and the same delay scenarios. In short, these counterexamples provide different execution patterns (instruction classes, baseline latencies, dependencies, etc.) leading to the same delay scenario, from the same initial stages. In both Tables 10.3a and 10.4a, the attributes of downstream instructions and upstream instructions are presented in pairs (i.e., the first element referring to the I-pipeline and the second element, to the LS-pipeline). Finally, the term for the initial upstream stage only concerns the I-pipeline, the LS-instruction being always initialized in the pre stage.

Table 10.3: Counterexamples found for the code-specific model configuration (TR9)—also see Table 10.2a.

(a) Broad-spectrum exploration (Algorithm 10.2)

	Initial conditions		Delay scenario		
multiplicity	$stage_{dw}$	$stage_{up.I}$	depth	$stage_{dw}$	$stage_{up}$
1	(IF, pre)	pre	3	(EX2, ID)	(ID, IF)
1	(EX, IF)	pre	7	(post, ID)	(IF, IF)
1	(post, IF)	pre	9	(post, ID)	(IF, IF)
1	(IF, pre)	pre	10	(post, ID)	(pre, IF)
16	(EX, ID)	ID	2	(WB, ID)	(EX, IF)
15	(ID, ID)	IF	3	(WB, ID)	(ID, IF)
1	(EX, ID)	pre	2	(WB, ID)	(IF, IF)
15	(ID, ID)	pre	3	(WB, ID)	(IF, IF)
13	(EX, ID)	IF	2	(WB, ID)	(IF, pre)

(b) Delay-scenario enumeration (Algorithm 10.3)

	Initial conditions		Delay scenario		
	$stage_{dw}$	$stage_{up.I}$	depth	$stage_{dw}$	$stage_{up}$
1	(IF, pre)	pre	7	(EX, ID)	(IF, IF)
2	(ID, ID)	IF	1	(EX, ID)	(IF, pre)
3	(IF, pre)	pre	6	(EX2, ID)	(IF, IF)
4	(IF, pre)	pre	3	(EX2, ID)	(IF, pre)
5	(post, pre)	pre	3	(post, ID)	(ID, IF)
6	(EX, IF)	pre	7	(post, ID)	(IF, IF)
7	(post, pre)	pre	3	(post, ID)	(IF, pre)
8	(IF, pre)	pre	7	(post, ID)	(pre, IF)
9	(EX, ID)	ID	2	(WB, ID)	(EX, IF)
10	(ID, ID)	IF	3	(WB, ID)	(ID, IF)
11	(IF, pre)	pre	10	(WB, ID)	(IF, IF)
12	(ID, ID)	IF	3	(WB, ID)	(IF, pre)
13	(IF, pre)	pre	9	(WB, ID)	(pre, IF)

Consistency of the Delay Scenarios. By definition of a delay scenario, at least one element (related to the LS or I-pipeline) of the $progress_{dw}$ term should be false, at least one of $delay_{dw}$ should have a positive value and no element of $delay_{dw}$ should be greater than 1. These conditions are checked at the end of the respective algorithms. All the counterexamples reported in Table 10.3a have the same values for the $class$ attribute ($(mac, load)$ for both the downstream and upstream tuples), fixed accordingly with the code-specific example (as in Sec. 9.2.2). We also check that the $memdep$ and $prio\_SB$ attributes are always false in the code-specific counterexamples, a condition that is not necessary for the general configuration.

Analysis of the Delay Scenarios. The delay scenarios in Table 10.3a correspond to the (LS) downstream instruction about to perform a data memory access while another memory access to fetch an upstream instruction is in-progress. Indeed, all have in common a delay scenario, as represented in Fig. 10.2c, with the LS downstream instruction waiting in the **ID** stage[6] and at least one upstream instruction in the **IF** stage. Moreover, we verify that the initial terms for the bus accesses are

[6]Just before the EX stage where the SRI bus access will occur through the DMI (Sec. 9.1.2).

Table 10.4: Counterexamples found for the general model configuration (TR6)—also see Table 10.2b.

(a) Broad-spectrum exploration (Algorithm 10.2)

	Initial conditions		Delay scenario		
multiplicity	$stage_{dw}$	$stage_{up.I}$	depth	$stage_{dw}$	$stage_{up}$
1	(IF, WB)	pre	1	(ID, WB)	(IF, IF)
1	(post, ID)	ST	3	(post, WB)	(ST, EX)
32	(pre, ID)	pre	1	(pre, EX)	(IF, IF)
2	(pre, EX)	pre	1	(pre, EX)	(IF, IF)
22	(pre, ID)	pre	1	(pre, EX)	(IF, IF)
1	(pre, EX)	pre	1	(pre, EX)	(IF, IF)
19	(pre, ID)	pre	1	(pre, EX)	(IF, IF)
2	(pre, ID)	pre	1	(pre, EX)	(pre, IF)
71	(pre, IF)	pre	1	(pre, ID)	(IF, IF)
14	(pre, IF)	pre	1	(pre, IF)	(pre, pre)
4	(pre, WB)	pre	1	(pre, post)	(IF, IF)
3	(pre, ST)	pre	1	(pre, ST)	(IF, IF)
3	(pre, WB)	pre	1	(pre, ST)	(IF, IF)
4	(pre, ST)	pre	1	(pre, ST)	(IF, IF)
2	(pre, WB)	pre	1	(pre, ST)	(IF, IF)
6	(pre, ST)	pre	1	(pre, ST)	(IF, IF)
3	(pre, WB)	pre	1	(pre, ST)	(IF, IF)
4	(pre, ST)	pre	1	(pre, ST)	(IF, IF)
6	(pre, WB)	pre	1	(pre, ST)	(IF, pre)
1	(pre, ST)	pre	1	(pre, ST)	(IF, pre)
3	(pre, WB)	pre	1	(pre, ST)	(IF, pre)
4	(pre, ST)	pre	1	(pre, ST)	(pre, IF)
2	(pre, WB)	pre	1	(pre, ST)	(pre, IF)
2	(pre, ST)	pre	1	(pre, ST)	(pre, pre)
6	(pre, EX)	pre	1	(pre, WB)	(IF, IF)

set accordingly.

Fig. 10.2 exemplifies such a delay scenario from the 16 gathered counterexamples presented in Table 10.3a. Some initial terms of these counterexamples, i.e., the pipeline stages for the dw and up instructions in both pipelines, their class, and the WAW dependencies, are represented in Fig. (10.2a). The subsequent unrolling of the transition system leads to an intermediate state (10.2b) (not exported by the algorithm), where the $dw.LS$ instruction experiences stalling because of a WAW dependency (see Sec. 9.1.4). Lastly, the next unrolling of the transition system, thus with a $depth$ of 2, leads to the delay scenario (10.2c) reported by the algorithm. All the other counterexamples presented in Tables 10.3 and 10.4 are interpreted in the same way.

The general configuration, presented in Table 10.4a, aims towards identifying patterns that do not exist in the code-specific configuration. Such patterns can occur when store instructions create new interference on the bus. For instance, the (6) counterexamples gathered in the last line of Table 10.4a show a delay scenario where a store instruction (i.e., $dw.LS$), is stalled in the WB stage because of the fetching of the upstream instructions, requiring the SRI bus with a higher priority.

Finally, recall that a delay scenario may be entailed by different initial conditions. Initial conditions may differ in various ways (e.g., combinations of baseline latencies and WAW dependencies) and thus lead to this scenario with the *same* depth. Here, in both core-model configurations, the counterexamples represented

Table 10.4

(b) Delay-scenario enumeration (Algorithm 10.3)

	Initial conditions				Delay scenario		
	$class_{dw}$	$class_{up}$	$stage_{dw}$	$stage_{up.I}$	depth	$stage_{dw}$	$stage_{up}$
1	(other_op, load_op)	(other_op, load_op)	(IF, pre)	pre	7	(EX, ID)	(IF, IF)
2	(other_op, store_op)	(mac_op, load_op)	(ID, WB)	IF	1	(EX, WB)	(ID, IF)
3	(other_op, store_op)	(other_op, load_op)	(pre, ID)	pre	3	(EX, WB)	(IF, ID)
4	(other_op, store_op)	(mac_op, load_op)	(ID, WB)	pre	1	(EX, WB)	(IF, IF)
5	(mac_op, store_op)	(other_op, load_op)	(IF, pre)	pre	9	(EX, WB)	(pre, IF)
6	(mac_op, load_op)	(other_op, load_op)	(IF, pre)	pre	10	(EX2, ID)	(IF, IF)
7	(mac_op, load_op)	(mac_op, load_op)	(EX, ID)	IF	1	(EX2, ID)	(IF, pre)
8	(mac_op, load_op)	(other_op, other_op)	(pre, IF)	pre	4	(EX2, ID)	(pre, IF)
9	(mac_op, store_op)	(mac_op, load_op)	(pre, IF)	pre	8	(EX2, WB)	(EX, ID)
10	(mac_op, store_op)	(mac_op, load_op)	(EX, WB)	ID	1	(EX2, WB)	(EX, IF)
11	(mac_op, store_op)	(mac_op, load_op)	(EX, WB)	IF	1	(EX2, WB)	(ID, IF)
12	(mac_op, store_op)	(mac_op, load_op)	(EX, WB)	pre	1	(EX2, WB)	(IF, IF)
13	(mac_op, store_op)	(other_op, other_op)	(IF, pre)	pre	9	(EX2, WB)	(pre, IF)
14	(other_op, store_op)	(other_op, load_op)	(pre, EX)	pre	2	(ID, WB)	(IF, ID)
15	(other_op, store_op)	(other_op, load_op)	(pre, IF)	pre	9	(ID, WB)	(IF, IF)
16	(mac_op, store_op)	(other_op, other_op)	(IF, pre)	pre	5	(ID, WB)	(pre, IF)
17	(other_op, store_op)	(other_op, load_op)	(pre, WB)	pre	1	(IF, WB)	(pre, IF)
18	(other_op, store_op)	(other_op, other_op)	(IF, WB)	pre	1	(IF, WB)	(pre, pre)
19	(mac_op, load_op)	(other_op, load_op)	(EX, IF)	IF	6	(post, ID)	(ID, IF)
20	(other_op, load_op)	(mac_op, other_op)	(EX, IF)	pre	6	(post, ID)	(IF, IF)
21	(other_op, load_op)	(other_op, load_op)	(EX, ID)	IF	2	(post, ID)	(IF, pre)
22	(other_op, load_op)	(other_op, other_op)	(post, pre)	post	3	(post, ID)	(post, IF)
23	(mac_op, load_op)	(mac_op, load_op)	(ID, IF)	pre	4	(post, ID)	(pre, IF)
24	(other_op, load_op)	(other_op, load_op)	(post, IF)	ST	2	(post, ID)	(ST, IF)
25	(other_op, load_op)	(other_op, load_op)	(post, IF)	EX	8	(post, ID)	(WB, IF)
26	(mac_op, store_op)	(mac_op, other_op)	(post, WB)	ID	1	(post, WB)	(EX, IF)
27	(other_op, store_op)	(mac_op, load_op)	(pre, IF)	pre	9	(post, WB)	(EX2, ID)
28	(mac_op, store_op)	(mac_op, load_op)	(post, WB)	EX	1	(post, WB)	(EX2, IF)
29	(other_op, store_op)	(other_op, other_op)	(post, pre)	IF	5	(post, WB)	(ID, IF)
30	(other_op, store_op)	(mac_op, load_op)	(post, WB)	pre	1	(post, WB)	(IF, IF)
31	(other_op, store_op)	(mac_op, load_op)	(post, WB)	IF	1	(post, WB)	(IF, pre)
32	(other_op, store_op)	(other_op, load_op)	(ID, IF)	IF	5	(post, WB)	(post, EX)
33	(other_op, store_op)	(mac_op, load_op)	(post, WB)	post	1	(post, WB)	(post, IF)
34	(other_op, store_op)	(other_op, load_op)	(IF, pre)	pre	12	(post, WB)	(pre, IF)
35	(other_op, store_op)	(other_op, load_op)	(post, ID)	ST	3	(post, WB)	(ST, EX)
36	(mac_op, store_op)	(mac_op, load_op)	(post, WB)	ST	1	(post, WB)	(ST, IF)
37	(other_op, store_op)	(mac_op, load_op)	(post, pre)	pre	5	(post, WB)	(WB, EX)
38	(mac_op, store_op)	(other_op, load_op)	(IF, pre)	pre	9	(post, WB)	(WB, ID)
39	(mac_op, store_op)	(other_op, load_op)	(post, WB)	EX	1	(post, WB)	(WB, IF)
40	(mac_op, load_op)	(other_op, load_op)	(pre, EX)	pre	1	(pre, EX)	(IF, IF)
41	(other_op, load_op)	(other_op, other_op)	(pre, EX)	pre	1	(pre, EX)	(pre, IF)
42	(mac_op, load_op)	(mac_op, load_op)	(pre, EX)	pre	1	(pre, EX)	(pre, pre)
43	(other_op, load_op)	(mac_op, store_op)	(pre, IF)	pre	1	(pre, ID)	(IF, IF)
44	(other_op, load_op)	(mac_op, other_op)	(pre, IF)	pre	1	(pre, ID)	(pre, IF)
45	(other_op, load_op)	(other_op, other_op)	(pre, IF)	pre	1	(pre, IF)	(pre, pre)
46	(other_op, other_op)	(other_op, load_op)	(pre, WB)	pre	1	(pre, post)	(IF, IF)
47	(other_op, other_op)	(other_op, load_op)	(pre, WB)	pre	1	(pre, post)	(pre, IF)
48	(mac_op, store_op)	(mac_op, load_op)	(pre, ST)	pre	1	(pre, ST)	(IF, IF)
49	(mac_op, store_op)	(mac_op, load_op)	(pre, ST)	pre	1	(pre, ST)	(IF, pre)
50	(other_op, store_op)	(mac_op, load_op)	(pre, ST)	pre	1	(pre, ST)	(pre, IF)
51	(mac_op, store_op)	(mac_op, load_op)	(pre, WB)	pre	1	(pre, ST)	(pre, pre)
52	(mac_op, store_op)	(mac_op, other_op)	(pre, WB)	pre	1	(pre, WB)	(IF, IF)
53	(other_op, store_op)	(other_op, load_op)	(pre, EX)	pre	1	(pre, WB)	(pre, IF)
54	(mac_op, load_op)	(mac_op, other_op)	(pre, IF)	pre	5	(WB, ID)	(IF, IF)
55	(mac_op, load_op)	(other_op, load_op)	(ID, ID)	IF	3	(WB, ID)	(IF, pre)
56	(mac_op, load_op)	(other_op, other_op)	(pre, IF)	pre	5	(WB, ID)	(pre, IF)
57	(other_op, store_op)	(mac_op, load_op)	(pre, IF)	pre	8	(WB, WB)	(EX, ID)
58	(other_op, store_op)	(mac_op, load_op)	(EX, WB)	ID	1	(WB, WB)	(EX, IF)
59	(mac_op, store_op)	(mac_op, load_op)	(pre, IF)	pre	9	(WB, WB)	(EX2, ID)
60	(other_op, store_op)	(mac_op, load_op)	(EX, WB)	IF	1	(WB, WB)	(ID, IF)
61	(other_op, store_op)	(mac_op, load_op)	(pre, IF)	pre	4	(WB, WB)	(IF, ID)
62	(other_op, store_op)	(mac_op, load_op)	(EX, WB)	pre	1	(WB, WB)	(IF, IF)
63	(other_op, store_op)	(mac_op, load_op)	(EX, WB)	IF	1	(WB, WB)	(IF, pre)
64	(mac_op, store_op)	(other_op, load_op)	(IF, IF)	pre	4	(WB, WB)	(pre, IF)

by one line always share the same depth for the delay scenario. It is important to note that the experimental results are inherent to how the solver finds the counterexamples.

10.3.2 . Analysis of the Delay-Scenario Enumeration

Presentation of the Results. Tables 10.3b and 10.4b also represent the complete results of the delay-scenario enumeration, through Algorithm 10.3, on both core-model configurations. The delay scenarios in both tables (i.e., from line 7 of Algorithm 10.3) are **inherently different from one another** and drive the state-space exploration (i.e., line 10 of the algorithm). For each counterexample, we report the depth of the delay scenario and the extended initial conditions. For the general core configuration, we also present the instruction classes.

Correction of the Model. The procedure for multiple counterexample generation provided valuable feedback wrt. the TriCore model, as it exposed an incorrect execution scenario in the model. More precisely, Table 10.4 presents execution snapshots where I-instructions are incorrectly found in the ST stage, since only the store instructions, in the LS-pipeline, may access this stage. This is shown, for instance, in the second counterexample of Table 10.4a in the $stage_{up.I}$ column. The incorrect execution scenario was due to a missing assumption, which we eventually added in the model. The possible values for the initial stage of the $up.I$ instruction are specified through $\mathrm{UCLID5}$ assumptions on the $dw.I$ instruction in the initial state, in order to respect the upstream/downstream pipeline order. However, when $dw.I$ was initialized in the $post$ stage, the missing assumption caused the $up.I$ stage to be initialized with *any* existing stage value, in a non-deterministic way (see Sec. 3.3.1). Such omissions increase the state-space size and hence, the number of false positives that are reported when analyzing the counterexamples (they do *not* entail false negatives).

Same Delay Scenarios with New Patterns. The delay scenarios shared by Tables 10.3b and 10.3a are from different counterexample lengths and, consequently, are not exposed by the same extended initial conditions.[7] For instance, the counterexample in line 11, in Table 10.3b, shows a delay scenario that is already present in Table 10.3a ($(WB, ID), (IF, IF)$) at different depths and under different initial conditions (e.g., baseline latencies or WAW attributes). This entails new patterns for the same delay scenario.

Equivalent Delay Scenarios. When a delay scenario is found by Algorithm 10.3, the combination of the four stages (exposed by the pipeline model) and positive delay value are to be eliminated through additional assertions over *each depth* (as in Sec. 10.2.2). However, not the four instructions are responsible for the delay scenario (as shown in Fig. 10.2). For example, the delay scenario in the first coun-

[7] Note that the converse is not true, as mentioned above.

terexample of Table 10.3a differs by the $stage_{up}$ column wrt. the counterexample in line 3 of Table 10.3b. More precisely, the only variable with a different value in the two delay scenarios is the $stage_{up.I}$ variable. This variable is not responsible for the delay, as explained above (it is independent of a load being delayed when an upstream instruction is fetched). As such, the delay scenario from Table 10.3a (with the $up.I$ instruction in ID) is the same *effective* scenario[8] as the one from Table 10.3b (with the $up.I$ instruction in IF), merely with a greater global progress in the pipeline.

Now, in the delay-scenario enumeration procedure, the four-stage combination of the delay scenario with $up.I$ in the ID stage (as in Table 10.3a) would be possible only in *prefixes* of new counterexamples (when the delay is still zero), not in suffixes. Yet, the delay scenario reported from the broad exploration (with $stage_{up.I}$ in ID) is included in the suffix of the delay scenario reported by the enumeration procedure (with $stage_{up.I}$ in IF, as in Table 10.3b), and hence, it is not reported as a new counterexample by the solver. As a final remark, the presentation order of these counterexamples does not reflect the order in which the exploration produces them.

New Delay Scenarios. The code-specific counterexamples in Table 10.3 show a single type of *effective* delay scenario—when the downstream load instruction is delayed by an upstream fetching. For general counterexamples, the results of the delay-scenario enumeration, in Table 10.4b, reveals delay scenarios that are not reported by the broad-spectrum exploration, shown in Table 10.4a. We identify in Table 10.4a (six) delay scenarios consisting in a downstream instruction stalled in the WB stage because of an upstream instruction in the IF stage (last line). We also need to identify and exclude impossible counterexamples. Table 10.4b shows three such counterexamples (at lines 24, 35 and 36) and a delay scenario involving the LS downstream instruction, stuck in the WB stage due to a fetching access of an upstream instruction (line 52). Several counterexamples from a *new* delay scenario, also involving the LS downstream instruction in the WB stage, are presented in lines 32 and 37 of Table 10.4b. In these cases, the downstream instruction is stalled because of a load upstream instruction performing a data memory access through the SRI bus (the upstream instruction has a higher priority as the store buffer is not full).

10.4 . Summary: a Step towards Accurate TA Patterns

In this chapter, we presented our counterexample-based heuristics to discover execution scenarios that can entail amplification TAs and we reported the results of their application to the TriCore model.

[8] Namely, the same scenario if we omit the pipeline stages that are not responsible for the delay scenario.

The broad-spectrum strategy is a heuristic designed to find multiple counterexamples efficiently. It allows uncovering TA patterns that result from various (extended) initial conditions. It could also be specialized to explore many initial conditions entailing a *specific* delay scenario. The delay-scenario-enumeration strategy is a heuristic designed to derive an exhaustive list of the delay scenarios. It requires further analyzing the initial conditions that can lead to these scenarios but provides *all* the delay scenarios that can manifest, and thus all the hardware sources of amplification TAs.

We explained that the verification strategies presented in this chapter require a more concrete model in order to determine precise execution patterns. As future work, we intend to apply these strategies on such a model, with a view to inserting (hardware or software) counter-measures that limit the occurrence of amplification TAs, e.g., through compilation instructions. This direction should imply further extensions to the current TriCore model, notably at the level of execution units and accurate formal ISA semantics.

The work developed in this chapter is not specific to TriCore, nor to *amplification* TAs. For instance, our heuristics could be applied to an SMT formulation of our OoO-pipeline specification (cf. Ch. 5). Note that SMT formulations could be derived from TLA^+ specifications (e.g., our pipeline specification: see Part III) through the use of a symbolic model checker [68] (cf. Sec. 3.3.2).

CONCLUSION & PROSPECTS

In this thesis, we studied *timing anomalies* (TAs), i.e., undesired phenomena that manifest at the level of two different execution traces for the same program path and that jeopardize *predictability* (cf. Part I). We distinguished two types of TAs, namely *counter-intuitive* TAs (a local slowdown yielding a larger global execution time) and *amplification* TAs (a local slowdown yielding a more significant global slowdown). We exemplified counter-intuitive TAs on the most documented class of hardware microarchitectures that allow TAs, specifically a simplified but representative Out-of-Order (OoO) pipeline template, and from the most current hardware source of variations, specifically the scheduling on the pipeline. We exemplified amplification TAs on in-order pipelines, which recent work showed to be prone to particular execution scenarios resulting from the interference on a shared memory bus.

We demonstrated that the various existing formal definitions of counter-intuitive TAs are limited in detecting TAs in practice, which explains the lack of tool support for the automatic detection of TAs (cf. Part II). We highlighted that the major issue shared by all these definitions resides in the lack of *causality*, in order to relate global effects to their local causes. We thus proposed a novel definition of counter-intuitive TAs, integrating the notion of causality (cf. Part III). This then allowed us to develop a detection procedure of counter-intuitive TAs and we showed that this procedure is able to detect TAs on our hardware formal model, and from a standard benchmark collection.

We also showed how the existing work on amplification TAs can be extended to handle a more complex microarchitecture than previously studied in this field, which requires scaling up an established verification procedure (cf. Part IV). We set up reductions that appropriately reduce the state space that must be explored. Moreover, contrary to the previously studied microarchitectures, our case study is not specifically designed to be predictable. Consequently, amplification TAs actually occur and it becomes necessary to identify the software patterns that entail them. We presented several strategies to monitor the covered state space, aiming at getting multiple execution scenarios that help building such patterns.

In ongoing work, we intend to improve our detection of counter-intuitive TAs, by refining both our OoO-pipeline model and our verification procedure. We intend to improve the model with a more concrete scheduler; the actual policy for assignments to functional units, notably, will be relevant when reasoning more in depth about *compositions*, a new, open problem that our work raised. Compositions also require additional information, linking variations to one another so as to represent side-effects on the hardware states. This might also help us integrate and address a related problem, that of *domino effects*, potentially caused by a chain of TAs.

We will also refine our model with other hardware resources (e.g., speculation mechanisms) that are known to be potential sources of TAs. We will thus need to adapt our procedure to integrate these hardware features into the input causality relationships. Besides, we intend to speed up our detection procedure, applied on the fly on the causal regions, which should serve as a basis to decompose the identification of TAs into subproblems, each of them concerns only a limited portion of the execution traces. From the refined procedure, we intend to allow for more variation sources (e.g., from the instruction cache) and to continue the practical assessment on benchmarks that we reported in this work.

We also intend to provide an accurate formal definition of amplification TAs, beyond the established precondition on which we based our work. Indeed, this property actually concerns a projection of amplification TAs on a single trace, whereas all TAs are hyperproperties referring to at least two distinct execution traces (cf. Ch. 6). Though this property enabled us to advance the systematic work on the detection of TAs, it does not allow for a better understanding of the *nature* of amplification TAs. Amplification TAs are usually defined in the wake of counter-intuitive TAs, and only illustrated from the scheduling of instructions on an OoO pipeline. The amplification TAs caused by contention of memory accesses, on which we focused, are sometimes introduced without a clear distinction from counter-intuitive TAs (cf. Ch. 4), thus making it impossible to properly analyze amplification TAs and their—distinct—consequences. We believe that our formal framework and the notion of causality introduced for counter-intuitive TAs will be also valuable for the purpose of clarifying amplification TAs, due to either OoO scheduling or contention. In particular, we have identified execution scenarios that could be TAs at first sight, but where the global time is not *caused* by the variation. We could also tackle the combined consequences of both classes of TAs: *In the proved presence of counter-intuitive* TA*s, could a static analysis be safe and reasonably precise without being exhaustive, if one can verify the absence of amplification* TA*s and insert appropriate penalties during the analysis?*

The precise identification of (any class of) TAs through automatic tools will be helpful in order to insert mitigation mechanisms and efficient counter-measures preserving convenient static analyses. In that regard, it is also of interest to integrate such formal pipeline models into a WCET analyzer. As a final remark, the ongoing development of open hardware initiatives might allow us to base our formal modeling approach on existing hardware descriptions, e.g., from HDL (Hardware Description Language) designs. This should be a further step towards detecting TAs over increasingly complex microarchitectures.

BIBLIOGRAPHY

[1] Benjamin Binder et al. “Formal Processor Modeling for Analyzing Safety and Security Properties”. In: *11th European Congress Embedded Real Time Systems (ERTS)*. 2022.

[2] Benjamin Binder et al. “Is This Still Normal? Putting Definitions of Timing Anomalies to the Test”. In: *IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2021, pp. 139–148. doi: 10.1109/RTCSA52859.2021.00024.

[3] Benjamin Binder et al. “The Role of Causality in a Formal Definition of Timing Anomalies”. In: *IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2022, pp. 91–102. doi: 10.1109/RTCSA55878.2022.00016.

[4] Benjamin Binder et al. “Scalable Detection of Amplification Timing Anomalies for the Superscalar TriCore Architecture”. In: *Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings*. Vol. 12327. Lecture Notes in Computer Science. Springer, 2020, pp. 151–169. doi: 10.1007/978-3-030-58298-2_6.

[5] Benjamin Binder et al. “Formal Modeling and Verification for Amplification Timing Anomalies in the Superscalar TriCore Architecture”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 24 (2022), pp. 415–440. issn: 1433-2787. doi: 10.1007/s10009-022-00655-1.

[6] Stefano Zanero. “Cyber-Physical Systems”. In: *Computer* 50.4 (2017), pp. 14–16.

[7] Alan Burns. “Scheduling Hard Real-Time Systems: a Review”. In: *Software Engineering Journal* 6.3 (1991), pp. 116–128.

[8] John A Stankovic. “Real-Time and Embedded Systems”. In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 205–208.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. isbn: 012383872X.

[10] Reinhard Wilhelm et al. “The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools”. In: *ACM Trans. Embed. Comput. Syst.* (May 2008). doi: 10.1145/1347375.1347389.

[11] Ingomar Wenzel et al. “Measurement-Based Timing Analysis”. In: *ISoLA*. 2008, pp. 430–444.

[12] Stephen Law and Ian Bate. “Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis”. In: *ECRTS*. ECRTS’16. 2016. doi: 10.1109/ECRTS.2016.21.

[13] Guillem Bernat, Antoine Colin, and Stefan M. Petters. “WCET Analysis of Probabilistic Hard Real-Time System”. In: *RTSS*. 2002, pp. 279–288.

[14] Francisco J. Cazorla et al. “Probabilistic Worst-Case Timing Analysis: Taxonomy and Comprehensive Survey”. In: *ACM Comput. Surv.* (2019). doi: 10.1145/3301283.

[15] Robert Davis and Liliana Cucu-Grosjean. “A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems”. In: *LITES* (2019). doi: 10.4230/LITES-v006-i001-a003.

[16] Reinhard Wilhelm et al. “Static Timing Analysis for Hard Real-Time Systems”. In: vol. 5944. Jan. 2010, pp. 3–22. isbn: 978-3-642-11318-5. doi: 10.1007/978-3-642-11319-2_3.

[17] Jan Reineke and Rathijit Sen. “Sound and Efficient WCET Analysis in the Presence of Timing Anomalies”. In: *WCET*. 2009.

[18] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. “Towards Compositionality in Execution Time Analysis: Definition and Challenges”. In: *SIGBED Rev.* 12.1 (Mar. 2015), pp. 28–36. doi: 10.1145/2752801.2752805. url: https://doi.org/10.1145/2752801.2752805.

[19] Sebastian Hahn, Michael Jacobs, and Jan Reineke. “Enabling Compositionality for Multicore Timing Analysis”. In: *RTNS*. RTNS ’16. Brest, France: Association for Computing Machinery, 2016, pp. 299–308. isbn: 9781450347877. doi: 10.1145/2997465.2997471. url: https://doi.org/10.1145/2997465.2997471.

[20] Thomas Lundqvist and Per Stenström. “Timing Anomalies in Dynamically Scheduled Microprocessors”. In: *Real-Time Systems Symposium*. RTSS’99. 1999. doi: 10.1109/REAL.1999.818824.

[21] I. Wenzel et al. “Principles of Timing Anomalies in Superscalar Processors”. In: *QSIC*. 2005. doi: 10.1109/QSIC.2005.49.

[22] Jan Reineke et al. “A Definition and Classification of Timing Anomalies”. In: *WCET*. WCET’06. 2006. doi: 10.4230/OASIcs.WCET.2006.671.

[23] R. Kirner, A. Kadlec, and P. Puschner. "Precise Worst-Case Execution Time Analysis for Processors with Timing Anomalies". In: *ECRTS*. ECRTS'09. July 2009. doi: 10.1109/ECRTS.2009.8.

[24] Gernot Gebhard. "Timing Anomalies Reloaded". In: *WCET*. WCET'10. 2010. doi: 10.4230/OASIcs.WCET.2010.1.

[25] S. Hahn and J. Reineke. "Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core". In: *RTSS*. 2018.

[26] Mathieu Jan et al. "Formal Semantics of Predictable Pipelines: a Comparative Study". In: *ASP-DAC*. 2020, pp. 103–108. doi: 10.1109/ASP-DAC47756.2020.9045351.

[27] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies". In: *SIAM Journal on Applied Mathematics* 17.2 (1969), pp. 416–429. issn: 00361399.

[28] B. Andersson and J. Jonsson. "Preemptive Multiprocessor Scheduling Anomalies". In: *Parallel and Distributed Processing Symposium*. 2002. doi: 10.1109/IPDPS.2002.1015483.

[29] A.K. Mok and Wing-Chi Poon. "Non-Preemptive Robustness under Reduced System Load". In: *Int. Real-Time Systems Symposium*. 2005, pp. 10–209. doi: 10.1109/RTSS.2005.31.

[30] Alan Burns and Sanjoy Baruah. "Sustainability in Real-time Scheduling". In: *JCSE* 2 (Mar. 2008), pp. 74–97. doi: 10.5626/JCSE.2008.2.1.074.

[31] Petros Voudouris, Per Stenström, and Risat Pathan. "Timing-Anomaly Free Dynamic Scheduling of Task-Based Parallel Applications". In: *Real-Time and Embedded Technology and Applications Symposium*. 2017, pp. 365–376. doi: 10.1109/RTAS.2017.2.

[32] *AURIX TC21x/TC22x/TC23x Family 32-Bit Single-Chip Microcontroller User's Manual*. Infineon Technologies AG. Dec. 2014.

[33] *TriCore 1 Pipeline Behaviour and Instruction Execution Timing*. AP32071. Infineon Technologies AG. June 2004.

[34] Mihail Asavoae, Belgacem Ben Hedia, and Mathieu Jan. "Formal Executable Models for Automatic Detection of Timing Anomalies". In: *WCET*. Ed. by Florian Brandner. 2018. doi: 10.4230/OASIcs.WCET.2018.2. url: http://drops.dagstuhl.de/opus/volltexte/2018/9748.

[35] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. "Modeling Out-of-Order Processors for WCET Analysis". In: *Real-Time Syst.* 34 (2006), pp. 195–227. doi: 10.1007/s11241-006-9205-5.

[36] Berkeley University of California. *The Register Files and Bypass Network of the RISCV-BOOM CPU*. url: https://docs.boom-core.org/en/latest/sections/reg-file-bypass-network.html (visited on 12/18/2020).

[37] Glenn Hinton et al. "The Microarchitecture of the Pentium 4 Processor". In: *Intel Technology Journal* (2001).

[38] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of R&D* 11.1 (1967), pp. 25–33. doi: 10.1147/rd.111.0025.

[39] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving, Second Edition*. USA: Dover Publications, Inc., 2015. isbn: 0486780821.

[40] Stephan Merz. "Model Checking: A Tutorial Overview". In: *Modeling and Verification of Parallel Processes: 4th Summer School, MOVEP 2000 Nantes, France, June 19–23, 2000 Revised Tutorial Lectures*. Ed. by Franck Cassez et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 3–38. isbn: 978-3-540-45510-3. doi: 10.1007/3-540-45510-8_1. url: https://doi.org/10.1007/3-540-45510-8_1.

[41] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. Jan. 2001. isbn: 978-0-262-03270-4.

[42] Edmund M. Clarke et al. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods in System Design* 19 (2001), pp. 7–34.

[43] Edmund Clarke et al. "Model Checking: Back and Forth between Hardware and Software". In: *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. Ed. by Bertrand Meyer and Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 251–255. isbn: 978-3-540-69149-5. doi: 10.1007/978-3-540-69149-5_27. url: https://doi.org/10.1007/978-3-540-69149-5_27.

[44] Kim G. Larsen, Paul Pettersson, and Wang Yi. "Uppaal in a Nutshell". In: *Int. J. Softw. Tools Technol. Transf.* 1.1–2 (Dec. 1997), pp. 134–152. issn: 1433-2779. doi: 10.1007/s100090050010. url: https://doi.org/10.1007/s100090050010.

[45] G.J. Holzmann. "The Model Checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295. doi: 10.1109/32.588521.

[46] Alessandro Cimatti et al. “NuSMV: a New Symbolic Model Verifier”. In: *International conference on computer aided verification*. Springer. 1999, pp. 495–499.

[47] Marta Kwiatkowska, Gethin Norman, and David Parker. “PRISM: Probabilistic Symbolic Model Checker”. In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2002, pp. 200–204.

[48] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[49] Didier Lime et al. “Romeo: a Parametric Model Checker for Petri Nets with Stopwatches”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2009, pp. 54–57.

[50] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)* (1977), pp. 46–57.

[51] Leslie Lamport. “What Good is Temporal Logic?” In: *IFIP congress*. Vol. 83. 1983, pp. 657–668.

[52] J.R. Burch et al. “Symbolic Model Checking: 10^{20} States and Beyond”. In: *Information and Computation* 98.2 (1992), pp. 142–170. issn: 0890-5401. doi: https://doi.org/10.1016/0890-5401(92)90017-A. url: https://www.sciencedirect.com/science/article/pii/089054019290017A.

[53] Kenneth L. McMillan. “Symbolic Model Checking: an Approach to the State Explosion Problem”. In: 1992.

[54] Jörg Bormann et al. “Model Checking in Industrial Hardware Design”. In: *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*. DAC ’95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 298–303. isbn: 0897917251. doi: 10.1145/217474.217545. url: https://doi.org/10.1145/217474.217545.

[55] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. “Verifying Temporal Properties of Sequential Machines without Building their State Diagrams”. In: *Computer-Aided Verification*. Ed. by Edmund M. Clarke and Robert P. Kurshan. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 23–32. isbn: 978-3-540-38394-9.

[56] Henrik Reif Andersen. "An Introduction to Binary Decision Diagrams". In: *Lecture notes, available online, IT University of Copenhagen* (1997), p. 5.

[57] Armin Biere et al. "Symbolic Model Checking Using SAT Procedures instead of BDDs". In: Jan. 1999, pp. 317–320. doi: 10.1109/DAC.1999.781333.

[58] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. "Symbolic Reachability Analysis Based on SAT-Solvers". In: *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*. TACAS '00. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 411–425. isbn: 3540672826.

[59] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Cham: Springer International Publishing, 2018, pp. 305–343. isbn: 978-3-319-10575-8. doi: 10.1007/978-3-319-10575-8_11. url: https://doi.org/10.1007/978-3-319-10575-8_11.

[60] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. isbn: 978-3-540-45047-4.

[61] Alessandro Abate et al. "Counterexample Guided Inductive Synthesis Modulo Theories". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 270–288.

[62] B. A. Brady, D. Holcomb, and S. A. Seshia. "Counterexample-Guided SMT-driven Optimal Buffer Sizing". In: *2011 Design, Automation Test in Europe*. 2011, pp. 1–6. doi: 10.1109/DATE.2011.5763058.

[63] S. A. Seshia and P. Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis and Learning". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2018, pp. 1–10. doi: 10.1109/MEMCOD.2018.8556946.

[64] Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. "Modeling and Verification of Out-of-Order Microprocessors in UCLID". In: *Formal Methods in Computer-Aided Design*. Ed. by Mark D. Aagaard and John W. O'Leary. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 142–159. isbn: 978-3-540-36126-8.

[65] Rajeev Alur and David L. Dill. "A Theory of Timed Automata". In: *Theoretical Computer Science* 126 (1994), pp. 183–235.

[66] Nikolaj Bjørner et al. "Programming Z3". In: *Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures*. Ed. by Jonathan P. Bowen, Zhiming Liu, and Zili Zhang. Cham: Springer International Publishing, 2019, pp. 148–201. isbn: 978-3-030-17601-3. doi: 10.1007/978-3-030-17601-3_4. url: https://doi.org/10.1007/978-3-030-17601-3_4.

[67] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[68] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. "TLA+ Model Checking Made Symbolic". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). doi: 10.1145/3360549. url: https://doi.org/10.1145/3360549.

[69] Kaustuv Chaudhuri et al. "A TLA+ Proof System". In: *Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA)*. Doha, Qatar, 2008. url: https://hal.inria.fr/inria-00338299.

[70] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent System Using Temporal Logic Specifications: A Practical Approach". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '83. Austin, Texas: Association for Computing Machinery, 1983, pp. 117–126. isbn: 0897910907. doi: 10.1145/567067.567080. url: https://doi.org/10.1145/567067.567080.

[71] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Wang Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 268–283. isbn: 978-3-540-45319-2.

[72] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2004, pp. 168–176.

[73] Bastian Schlich and Stefan Kowalewski. "Model Checking C Source Code for Embedded Systems". In: *STTT* 11 (July 2009), pp. 187–202. doi: 10.1007/s10009-009-0106-5.

[74] E. Clarke and B. Mishra. "Automatic Verification of Asynchronous Circuits". In: *Logics of Programs*. Ed. by Edmund Clarke and Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 101–115. isbn: 978-3-540-38775-6.

[75] Jerry R. Burch and David L. Dill. "Automatic Verification of Pipelined Microprocessor Control". In: *Computer Aided Verification*. Ed. by David L. Dill. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 68–80. isbn: 978-3-540-48469-1.

[76] Jerry Burch. "Techniques for Verifying Superscalar Microprocessors". In: July 1996, pp. 552–557. isbn: 0-7803-3294-6. doi: 10.1109/DAC.1996.545637.

[77] Jens Ulrik Skakkebæk, Robert B. Jones, and David L. Dill. "Formal Verification of Out-of-Order Execution Using Incremental Flushing". In: *CAV*. 1998.

[78] Sergey Berezin et al. "Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-Order Processor Verification". In: *Formal Methods in Computer-Aided Design*. Ed. by Ganesh Gopalakrishnan and Phillip Windley. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 369–386.

[79] J. Eisinger et al. "Automatic Identification of Timing Anomalies for Cycle-Accurate Worst-Case Execution Time Analysis". In: *DDECS*. 2006. doi: 10.1109/DDECS.2006.1649563.

[80] Raimund Kirner, Albrecht Kadlec, and Peter Puschner. *Worst-Case Execution Time Analysis for Processors showing Timing Anomalies*. Tech. rep. TU Wien, 2009.

[81] Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. "What is a Timing Anomaly?" In: *WCET*. Vol. 23. WCET'12. 2012. doi: 10.4230/OASIcs.WCET.2012.1.

[82] Christine Rochange and Pascal Sainrat. "A Time-Predictable Execution Mode for Superscalar Pipelines with Instruction Prescheduling". In: Jan. 2005, pp. 307–314. doi: 10.1145/1062261.1062312.

[83] Jack Whitham and Neil Audsley. "Time-Predictable Out-of-Order Execution for Hard Real-Time Systems". In: *IEEE Transactions on Computers* 59.9 (2010), pp. 1210–1223. doi: 10.1109/TC.2010.109.

[84] Isaac Liu, Jan Reineke, and Edward A. Lee. "A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties". In: *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*. 2010, pp. 2111–2115. doi: 10.1109/ACSSC.2010.5757922.

[85] Isaac Liu et al. “A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance”. In: *IEEE 30th International Conference on Computer Design (ICCD)* (2012), pp. 87–93.

[86] Michael Zimmer et al. “FlexPRET: a Processor Platform for Mixed-Criticality Systems”. In: *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2014), pp. 101–110.

[87] Martin Schoeberl et al. “Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach”. In: *Proc. of Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. 2011.

[88] Alban Gruin et al. “Speculative Execution and Timing Predictability in an Open Source RISC-V Core”. In: *Real-Time Systems Symposium*. IEEE, 2021, pp. 393–404. doi: 10.1109/RTSS52674.2021.00043.

[89] Michael Platzer and Peter Puschner. “Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation”. In: *Euromicro Conference on Real-Time Systems*. Vol. 196. LIPIcs, 2021. isbn: 978-3-95977-192-4. doi: 10.4230/LIPIcs.ECRTS.2021.1.

[90] Benoît Dupont de Dinechin et al. “Time-Critical Computing on a Single-Chip Massively Parallel Processor”. In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2014, pp. 1–6. doi: 10.7873/DATE.2014.110.

[91] Alexander Metzner. “Why Model Checking Can Improve WCET Analysis”. In: *International Conference on Computer Aided Verification*. Springer. 2004, pp. 334–347.

[92] Benedikt Huber and Martin Schoeberl. “Comparison of Implicit Path Enumeration and Model-Checking-based WCET Analysis”. In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2009.

[93] Andreas Dalsgaard et al. “METAMOC: Modular Execution Time Analysis Using Model Checking”. In: vol. 15. Jan. 2010, pp. 113–123. doi: 10.4230/OASIcs.WCET.2010.113.

[94] Andreas Gustavsson et al. “Towards WCET Analysis of Multicore Architectures Using UPPAAL”. In: vol. 15. Jan. 2010, pp. 101–112. doi: 10.4230/OASIcs.WCET.2010.101.

[95] Ravindra Metta et al. “TIC: a Scalable Model-Checking-based Approach to WCET Estimation”. In: *ACM SIGPLAN Notices* 51.5 (2016), pp. 72–81.

[96] Mihail Asavoae, Mathieu Jan, and Belgacem Ben Hedia. “Formal Modeling and Verification for Timing Predictability”. In: *ERTS*. 2020.

[97] Zhenyu Bai et al. “Improving the Performance of WCET Analysis in the Presence of Variable Latencies”. In: *LCTES*. 2020, pp. 119–130. doi: 10.1145/3372799.3394371.

[98] Viet anh Nguyen et al. “Using Model Checking to Identify Timing Interferences on Multicore Processors”. In: *ERTS 2020 - 10th European Congress on Embedded Real Time Software and Systems*. Toulouse, France, 2020, pp. 1–10.

[99] Stephan Wilhelm and Björn Wachter. “Towards Symbolic State Traversal for Efficient WCET Analysis of Abstract Pipeline and Cache Models”. In: *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*. 2007.

[100] Theo Ungerer et al. “Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability”. In: *IEEE Micro* 30.5 (2010), pp. 66–75.

[101] Wei-Tsun Sun, Eric Jenn, and Hugues Cassé. “Build Your Own Static WCET Analyser: the Case of the Automotive Processor AURIX TC275”. In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. Jan. 2020.

[102] Clément Ballabriga et al. “OTAWA: An Open Toolbox for Adaptive WCET Analysis”. In: *SEUS 2010*. 2010, pp. 35–46.

[103] Michael R. Clarkson and Fred B. Schneider. “Hyperproperties”. In: *21st IEEE Computer Security Foundations Symposium*. 2008, pp. 51–65. doi: 10.1109/CSF.2008.7.

[104] G. Barthe, P.R. D’Argenio, and T. Rezk. “Secure Information Flow by Self-Composition”. In: *Proceedings. 17th IEEE Computer Security Foundations Workshop.* 2004, pp. 100–114. doi: 10.1109/CSFW.2004.1310735.

[105] Heiko Falk et al. “TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research”. In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Ed. by Martin Schoeberl. Vol. 55. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, 2:1–2:10.

[106] Michael Platzer and Peter Puschner. “A Processor Extension for Time-Predictable Code Execution”. In: *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*. 2021, pp. 34–42. doi: 10.1109/ISORC52013.2021.00016.

[107] *Architecture Overview Handbook, TriCore 1.3, 32-bit Unified Processor Core, IP Cores*. Infineon Technologies AG. May 2002.

[108] *Simulator for TriCore*. Lauterbach GmbH. Apr. 2021.

Thèse de doctorat de Benjamin BINDER, soutenue à Paris-Saclay, le 13 décembre 2022.
Benjamin Binder's PhD thesis, defended in Paris-Saclay, on December 13, 2022.

Definitions and Detection Procedures of Timing Anomalies for the Formal Verification of Predictability in Real-Time Systems

Abstract: The timing behavior of real-time systems is often validated through *timing analyses*, which are yet jeopardized by *timing anomalies* (TAs). A *counter-intuitive* TA manifests when a local speedup eventually leads to a global slowdown, and an *amplification* TA, when a local slowdown leads to an even larger global slowdown.

While counter-intuitive TAs threaten the soundness/scalability of timing analyses, tools to systematically detect them do not exist. We set up a unified formal framework for systematically assessing the definitions of TAs, concluding the lack of a practical definition, mainly due to the absence of relations between local and global timing effects. We address these relations through the *causality*, which we further use to revise the formalization of these TAs. We also propose a specialized instance of the notions for out-of-order pipelines. We evaluate our subsequent detection procedure on illustrative examples and standard benchmarks, showing that it allows accurately capturing TAs.

The complexity of the systems demands that their timing analyses be able to cope with the large resulting state space. A solution is to perform *compositional analyses*, specifically threatened by amplification TAs. We advance their study by showing how a specialized abstraction can be adapted for an industrial processor, by modeling the timing-relevant features of such a hardware with appropriate reductions. We also illustrate from this class of TAs how verification strategies can be used towards the obtainment of TA patterns.