Parallelism and timing predictability in real-time systems

Thomas Carle

MÉMOIRE

En vue de l'obtention de l'

HABILITATION À DIRIGER LES RECHERCHES

Délivrée par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *6/6/2024* **par :**

Thomas Carle

Parallelism and timing predictability in real-time systems

JURY

Pr. Christine Rochange	Marraine
Dr. HDR Mathieu Jan	Rapporteur
Pr. Isabelle Puaut	Rapporteure
Pr. Jan Reineke	Rapporteur
Dr. HDR Claire Pagetti	Examinatrice

École doctorale et spécialité :

MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de Recherche :

Institut de Recherche en Informatique de Toulouse (UMR 5505)

Remerciements

Les résultats relatés dans ce manuscrit sont le fruit de huit années de travaux collaboratifs avec mes collègues de l'IRIT et mes étudiants. Je souhaite ici les remercier pour tout le travail qu'ils ont fourni à mes côtés, toutes les discussions scientifiques que nous avons pu avoir et qui ont enrichi ma culture et ma vision du domaine.

Je tiens à remercier sincèrement les membres de mon jury, à commencer par mes trois rapporteurs pour le travail qu'ils ont fourni. Je voudrais en particulier remercier Isabelle Puaut, qui fut rapportrice de ma thèse de doctorat avant de participer à mon jury de recrutement à Toulouse. Je vois en sa participation à mon jury d'HDR comme un trait d'union sur le début de ma carrière, qui va se poursuivre par des collaborations que nous espérons fructueuses.

J'aimerais également remercier Claire Pagetti, et mes amis de l'ONERA, pour m'avoir permis de collaborer avec eux. Au delà de la qualité et de la productivité de nos collaborations, je les remercie de leur bonne humeur et de leur accueil chaleureux.

Enfin, je souhaite surtout remercier Christine Rochange, sans qui cette habilitation n'aurait pas été possible. Je ne saurais énumérer ici tout ce pourquoi je lui suis reconnaissant, depuis l'accueil qu'elle m'a réservé dans l'équipe, la manière dont elle m'a épaulé et guidé dans la construction de mon début de carrière, ses encouragements à me lancer sur mes orientations de recherche, sa présence bienveillante au quotidien, et tant d'autres choses. Nous avons en commun un niveau d'exigence élevé dans la qualité de nos productions scientifiques, mais également un sens de l'humour à toute épreuve, qui rend la vie au labo tellement agréable.

Contents

Chapter 1

General Introduction

I was recruited in Toulouse in September 2016, and I joined the TRACES team that specializes in worst-case execution time (WCET) analysis, and is responsible for the development and support of the OTAWA WCET analyser.

Since my arrival, I have been working on the topic of timing predictability for real-time systems with a focus on issues related to parallelism. This encompasses a wide panel of activities ranging from the static analysis of GPU software, to the modelling of caches behavior in highly optimized applications, the automatic generation of timing-predictable code, the interference-aware static scheduling of applications on multi-core processors, and the architectural design of predictable core acceleration mechanisms. This manuscript presents the main contributions I have been involved in, either as principal researcher or as a PhD thesis director, related to timing predictability in multi-core architectures.

1.1 Context

Real-time systems are computer systems whose correct operation depends on the respect of timing constraints. Classical examples include the control system of a plane, the security systems of an automatic subway or more recently the decision and control systems of an autonomous vehicle. In order to guarantee the respect of their timing constraints, the most critical real-time systems undergo static WCET analysis: each task composing the system is analyzed in order to derive a safe upper bound of its execution time on the hardware target. Then, the schedulability of the system is assessed by composing the bounds of each task using a schedulability criterion or a worst-case response-time analysis. Alternatively, the timing constraints can be checked as a static schedule is constructed. Obtaining safe and tight bounds requires a precise model of the temporal behavior of the hardware target. As the hardware gets more complex, the models and analyses must be adapted to provide a compromise between complexity of analysis and precision of the results. Parallelism has been progressively introduced in microprocessors as a way to increase their performance, starting with pipeline architectures in the 1980's. Since the 2000's, multi-core processors have been introduced as a way to overcome power dissipation, memory latency and instruction level parallelism bottlenecks, and have since become ubiquitous. The adoption of these architectural paradigms in processors used to implement real-time systems has been driven mainly by cost reduction and performance requirements. However, they incurred new and complex challenges to the timing predictability of real-time systems. The underlying theoretical problem that we explore in this manuscript can be summarised as follows: two (or more) logical entities being processed in parallel by separate hardware components contend for the access to a shared, sequential, non-pipelined, multi-cycles resource, and this contention jeopardizes the timing predictability of the overall system. This problem was investigated at two different

abstraction levels.

1.2 Contributions and plan

1.2.1 Design of predictable and efficient pipeline architectures

Following this introduction, Chapter 2 deals with the first instance of the problem, which is related to instruction level parallelism inside processor cores: two instructions being processed in separate pipeline stages can contend for a component such as a memory bus or a functional unit. Depending on the context, this contention can lead to a so-called timing anomaly: a situation in which it becomes hard to track the timing behavior of the analyzed (software+hardware) system. Most current Commercial Off The Shelf (COTS) hardware systems are considered vulnerable to timing anomalies. These anomalies make the static WCET analysis harder for single-core architectures, and preclude the use of efficient interference analysis methods in multi-core architectures. We tackled the problem by following the philosophy initiated by Hahn et al. in [40]: designing core pipelines in which the instructions are guaranteed to progress in a monotonous fashion. In Layman's terms, monotonic progress means that an instruction cannot be stalled anywhere in the pipeline by another instruction that appears later in the program order (and thus enters the pipeline later). This property guarantees the absence of timing anomalies, and is easier to prove formally than directly proving the freedom from timing anomalies. Hahn et al. have demonstrated this approach on a simple five-stage in-order pipeline: the pipeline was first modified to make the progress of instructions monotonous, then its behavior was modelled using a formal framework based on first-order logic, and finally using this model the monotonic behavior of the pipeline was formally proven. We applied this approach to a more complex core that features two acceleration mechanisms that were not handled in the work of Hahn et al.: speculative execution (branch predictors and return address stack) and store buffers. Simply turning off these mechanisms, as suggested by Hahn et al. is not acceptable in terms of performance, so the objective was to enable them in a way that enforces a monotonic progress while not degrading the core performance. We extended the formal framework in order to model the core, and provided original proofs for the monotonicity of the core with its acceleration mechanisms. These proofs were first conducted by hand, and have since been validated using the Coq proof assistant.

1.2.2 Multi-core timing analysis with the multi-phase model

Chapter 3 deals with the second instance of the problem, related to task level parallelism in multi-core architectures: two tasks being processed in parallel on separate cores can contend for a shared resource such as a memory bus or a memory controller. When this happens, one of the tasks accesses the resource, while the other is stalled. This phenomenon is referred to as timing interference. Depending on the architecture and the application, the effect of interference can significantly reduce the performance of the system. Additionally, for real-time systems, a worst-case response time analysis must be performed to ensure that the real-time constraints will always be met. This analysis must conservatively account for the effects of interference. Now, the abstraction gap between the interference analysis (that happens at task system level) and the source of the interference (memory instructions) incurs a significant over-estimation of the effects of interference, that may hinder the results of the Worst-Case Response Time (WCRT) analysis. We tackled this problem by proposing a finer grained abstraction: the execution of each task is represented by a sequence of temporal phases, called a multi-phase profile, that bounds the timing windows in which memory accesses can occur, thus providing more precise information for the interference analysis. This abstraction allows to significantly reduce the overestimation of the interference effects by sticking more closely to the actual memory access profiles, and

possibly to reduce the measured interference level by constructing and enforcing schedules in which the phases that perform the most accesses are not scheduled in parallel. In the past 7 years, we built a complete framework around the multi-phase model, composed of three major elements: a formal representation of the multi-phase model along with formal criteria for its correct implementation, an analysis method to obtain multi-phase representations for task systems, and scheduling heuristics that target the multi-phase model. This framework is described in details in the manuscript. As stated above, efficient interference analysis requires the absence of timing anomalies in the hardware. This is also true for the multi-phase method that we propose, so the two solutions that we present in the manuscript complement each other.

1.2.3 Conclusion

Finally, Chapter 4 summarizes the results presented in the manuscript, as well as the other research activities I conducted, and presents my research project.

1.3 Regarding this manuscript

Regarding the organization of the manuscript:

- each chapter starts by a quick introduction and state-of-the art that aim at defining the problem at hand and at introducing the terms and notions that will be used throughout the chapter,
- most of the proofs were omitted for readability. They are all available in the original publications that introduced the corresponding theorems.

Chapter 2

Designing provably predictable and efficient pipeline architectures

2.1 Introduction

The ever-growing performance requirements of embedded real-time systems lead to implement them on multi-core platforms. These platforms include several cores (which may feature out-of-order execution, dynamic branch prediction, speculative execution, private L1 caches) that share resources such as L2 and L3 cache memories, or the memory bus. However, their complexity challenges the analysis of execution and response times, which is required to schedule tasks in such a way that they all meet their deadlines (real-time constraints). The difficulty comes from the fact that tasks running simultaneously on different cores compete to access shared resources, which engenders delays that must be taken into account within the timing analysis. To cope with the explosion of the number of possible execution scenarii where co-running tasks generate interleaved accesses to shared resources, it is now commonly admitted that a *compositional* approach [39] that decouples the analyses of intra- and inter-core behaviors, i.e. execution time in isolation on the one hand, and delays induced by task interference on the other hand, is desirable. Estimating delays due to interference means, for example, upper bounding the demands of tasks to shared resources so as to estimate the amount of delay a co-running task can suffer. These delays can then be added to the local worst-case execution time that is evaluated assuming the task is running in isolation.

However, when execution cores are complex, this approach might not be valid. This is due to so-called *timing anomalies* [65]: a local worst case situation (e.g. a cache miss) does not necessarily lead to the global worst case (that is the worst-case execution time of the task under analysis), or a local delay (e.g. to due a cache miss) of a certain duration may result in a larger increase of the WCET. Timing anomalies are caused by the concurrent sharing of sequential resources by multiple components of a core, and can lead to instruction reordering within the pipeline. The consequence of the risk of timing anomalies is that any additional delay due to task interference should be precisely identified: the instruction impacted by the delay should be known. This does not fit compositional approaches that, instead of considering every possible interleaving of tasks accessing a shared resource (which is intractable due to the huge number of possibilities), have a global view of the amount of conflicts and of the total resulting delay. To summarize, timing anomalies are a serious obstacle to the implementation of compositional approaches and question the feasibility of accurate timing analysis for multicore-based real-time systems.

To overcome these difficulties, the strictly in-order (SIC) core [40] approach proposes (i) structural modifications that suppress the risk of timing anomalies in an in-order processor design and (ii) a

modelling framework to formally prove the good timing properties of the modified design. The key idea in this approach is to impose a strict execution order in which the progression of any instruction in the pipeline depends only on how the previous instructions in the code have already progressed. In-order pipelines that enforce this property and do not implement speculative execution are proven to be free of timing anomalies and timing compositional: considering only the local worst cases leads to a safe WCET, and delays due to multi-core interference can be statically bounded and safely added to the WCET of the interfering tasks. This allows trading off between the precision and efficiency of the WCET analysis while keeping its outcome sound. The SIC core is about 7% slower than the original core.

In this chapter we present the solutions that we developed in order to leverage this approach and its formal framework to a more complex core with a higher baseline performance than the one used in [40]: the open source RISC-V Ariane [78] core, which implements the RISC-V instruction set and features some advanced mechanisms such as dynamic branch predictors and multiple functional units that allow instruction parallelism. We call our modified core the Mostly IN-Order Timing predictAble pRocessor: MINOTAuR.

The key contributions are the following:

- we provide a formal model of the MINOTAuR core obtained by applying some restrictions from the SIC on the Ariane core, while keeping features such as branch prediction. We prove its timing predictability and we evaluate its performance on an FPGA: the loss is less than 2% compared to Ariane.
- we introduce a design extension for caches and return address stacks to support timing predictable speculative execution.
- we introduce a generic gating mechanism to prevent concurrency between the data cache and the store buffer for accesses to the memory bus, and apply it to MINOTAuR.

In this introduction we start by presenting the state of the art regarding timing predictable processors, and then briefly describe the original Ariane processor as well as the Ariane$^+$ core that served as a starting point and a baseline for our work.

2.1.1 Related work

2.1.1.1 Timing predictability

A processor is said *timing predictable* when there are no timing anomalies and it is timing compositional [40].

A timing anomaly occurs when a shorter latency for one instruction in a sequence (e.g. a cache hit instead of a cache miss for a load instruction) counter-intuitively makes the execution time of the sequence longer, or when a longer latency for one instruction leads to an even longer increase for the total execution time of the sequence [75, 52, 11]. This makes the timing analysis more complex since all the possible situations have to be considered. Several authors have investigated this, putting forward several definitions and means to detect whether a processor is prone to such timing anomalies [4, 47, 28, 30, 65, 12]. It turns out that most of off-the-shelves cores, even the simplest ones, may suffer from timing anomalies. This motivates the design of timing-anomalies-free processors (see Section 2.1.1.2).

Timing compositionality simplifies the timing analysis of a multi-core system [43]. It avoids a very complex fully-integrated system analysis in favor of a combination of analyses of individual components. An approach to sound and precise compositional timing analysis for multicore systems is proposed in [39].

2.1.1.2 Timing predictable processor architectures

Several ways have been considered to favor timing predictability in hardware platforms [58, 5, 59].

The Kalray MPPA-256 processor [25] has been designed with timing predictability in mind. In addition to its VLIW architecture (initially motivated by energy considerations), architectural features are supposed to fit the capabilities of WCET analysis: LRU-replacement caches, in-order execution, prevention of pipeline hazards, and absence of branch prediction.

PTARM [50] is an implementation of a precision-timed (PRET) machine [51]. It employs a repeatable thread-interleaved pipeline. Timing predictability is achieved at the cost of degraded performance for individual threads, while the instruction throughput is maintained over the set of active threads.

Patmos [69] features a statically-scheduled (VLIW) dual-issue pipeline and specific timing analysable caches, such as the method and stack caches. It has been used to build a real-time-aware multicore system in the T-CREST project [70]. Although it has been designed to be timing predictable, this has not been formally proven to the best of our knowledge.

In [40, 41], Hahn and Reineke introduce SIC, a strictly in-order core, and show that it is free of timing anomalies and timing compositional. Their formal framework used to prove these two properties is summarized in Section 2.1.1.3. SIC is a simple 5-stage in-order pipelined processor in which the instruction fetch is gated in order to guarantee that an instruction can never be delayed by a younger instruction.

2.1.1.3 A formal framework to prove timing predictability

A framework to express the concrete semantics of a processor pipeline is proposed in [42]. It relies on the concept of *progress* of an instruction within the pipeline, defined as the pipeline stage the instruction resides in and the number of cycles remaining to complete the stage. If $\mathcal{S}$ is the set of pipeline stages, the progress of an instruction belongs to $\mathcal{P} := \mathcal{S} \times \mathbb{N}_0$. A pipeline state can then be described by the subset $\mathcal{C} \subseteq \mathcal{I} \to \mathcal{P}$, where $\mathcal{I}$ is the sequence of executed instructions. With a partial order $\sqsubset_{\mathcal{S}}$ on $\mathcal{S}$, it is possible to define an order $\sqsubseteq_{\mathcal{P}}$ on $\mathcal{P}$:

Definition 1: Progress order

$$\forall (s_a, n_a), (s_b, n_b) \in \mathcal{P}, (s_a, n_a) \sqsubseteq_{\mathcal{P}} (s_b, n_b) :\Leftrightarrow s_a \sqsubset_{\mathcal{S}} s_b \vee (s_a = s_b \wedge n_a \geq n_b)$$

Considering the execution of a given sequence of instructions $\mathcal{I}$, pipeline state c_b has at least the progress of c_a if every instruction in $\mathcal{I}$ has a better (or same) progress in c_b than in c_a :

Definition 2: Pipeline state order

$$c_a \sqsubseteq c_b :\Leftrightarrow \forall i \in \mathcal{I} \,.\, c_a(i) \sqsubseteq_{\mathcal{P}} c_b(i)$$

where $c(i)$ denotes the progress of instruction i in state c.

The behaviour of the pipeline is specified by the function $cycle : \mathcal{C} \to \mathcal{C}$ that relates a pipeline state to its successor.

In [40], this framework is used to model the behaviour of the SIC pipeline. The progress of an instruction i after one clock cycle is specified as a function of the current pipeline state c: the instruction may remain in its current stage or advance to the next stage ($s = c.nstg(i)$) when it is ready to ($c.ready(i)$) and if that stage is clear of any previous instruction ($c.free(s)$)

Based on this model, the authors prove the following major property for the SIC processor.

Property 1: *Update Enable*

Let c_a and c_b be two pipeline states, $i \in \mathcal{I}$ be an instruction with equal progress in c_a and c_b ($c_a(i) = c_b(i)$), and all instructions $j < i$ have progressed more in c_b than c_a ($c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$). If i advances to the next pipeline stage in c_a, it advances in c_b as well:

$$\begin{cases} c_a.ready(i) \Rightarrow c_b.ready(i) \\ c_a.free(c_a.nstg(i)) \Rightarrow c_b.free(c_b.nstg(i)) \end{cases}$$

Several lemmas and theorems follow from this sole property and are thus valid for any processor that meets the property. We reformulate them below to reflect that. Proofs can be found in [40].

Lemma 1: *Progress Dependence*

When Property 1 holds, the progress of an instruction i only depends on the progress of previous instructions (and never on the progress of subsequent instructions):

$$\forall c_a, c_b \in \mathcal{C} : [\forall i : (\forall j \leq i : c_a(j) = c_b(j)) \Rightarrow cycle(c_a)(i) = cycle(c_b)(i)]$$

Lemma 2: *Positive Progress*

When Property 1 holds, the successor of a pipeline state c has more progress than c:

$$\forall c \in \mathcal{C} \ : \ c \sqsubset cycle(c)$$

where $\forall c_a, c_b \in \mathcal{C}, c_a \sqsubset c_b \Leftrightarrow c_a \sqsubseteq c_b \wedge \neg(c_b \sqsubseteq c_a)$

This formulation is a generalization of Lemma 2 in [40] to any in-order pipeline that enforces Property 1. The proof arguments of [40] hold in this more general context.

Theorem 1: *Monotonicity*

The cycle behavior of a processor that satisfies Property 1 is monotonic:

$$\forall c_a, c_b \in \mathcal{C} : c_a \sqsubseteq c_b \Rightarrow cycle(c_a) \sqsubseteq cycle(c_b)$$

Theorem 2

Let $i \in \mathcal{I}$ be an arbitrary instruction, and pipeline states $c_a, c_b \in \mathcal{C}$ be such that $c_a \sqsubseteq c_b$. Then:

$$f(c_a, i) \geq f(c_b, i)$$

where $f(c, i)$ is the finish time of instruction i starting from pipeline state c recursively defined as:

$$f(c, i) := \begin{cases} 0 & : c(i) = (post, 0) \\ 1 + f(cycle(c), i) & : otherwise \end{cases}$$

with *post* being a fictive pipeline stage that contains all the instructions that have left the pipeline.

Following these theorems, the authors of [40] demonstrate that the SIC processor is free of timing anomalies with respect to uncertain cache behaviour, and timing-compositional with respect to un-

certain cache behaviour and uncertain latency to the main memory. Uncertainties are reflected in the processor model by:

- $ichit(i)$ (resp. $dchit(i)$): true if instruction i results in an instruction (resp. data) cache hit
- $memlat_{f/d}$: memory latency in case of an instruction (resp. data) cache miss for instruction i

Theorem 3: ***Anomaly freedom with respect to cache uncertainty***

Let two valuations of $dchit$ (or $ichit$) be given that differ for an arbitrary instruction $i \in \mathcal{I}$. The valuation that predicts a cache miss, i.e. the local worst case, will lead to a finishing time at least as high as the valuation that predicts a cache hit, i.e. the local best case.

Theorem 4: ***Compositionality with respect to latency prolongation***

Let two valuations of $memlat_d$ (or $memlat_f$) be given that differ by p cycles for an arbitrary instruction $i \in \mathcal{I}$, e.g. due to shared bus blocking. The valuation that predicts a longer latency leads to a finishing time at most p cycles higher than the valuation that predicts the shorter latency.

The proof does not depend on the processor (provided it fulfills Property 1) and is given in [40].

Theorem 5: ***Compositionality with respect to cache uncertainty***

Let two valuations of $dchit$ (or $ichit$) be given that differ for an arbitrary instruction $i \in \mathcal{I}$. The valuation that predicts a cache miss will lead to a finishing time at most p cycles higher than the valuation that predicts a cache hit. For the SIC processor, p is twice the memory latency for a data cache miss with a write-through policy and five times the memory latency for an instruction cache miss.

The proof given in [40] is specific to the SIC processor.

2.1.2 Hardware state buffering mechanisms

In Section 2.2.3 we present hardware mechanisms to save the state of the Return Address Stack and of the instruction cache during speculative execution. These mechanisms rely on backup copies that are later committed or discarded when the corresponding branch instruction is resolved. Similar mechanisms were suggested in [44] but, to the best of our knowledge, they were not implemented. InvisiSpec [77] is an alternative mechanism that stores speculative loads in a buffer, but is suited for data caches in out-of-order processors, committing the loaded values step by step after each load. Our solution targets instruction caches instead, and protects the age of the cache blocks as well as their contents. Moreover, in our solution, commits occur only when speculative branches are resolved.

2.1.3 A baseline RISC-V core

Our baseline core is a slightly modified version of the Ariane core [78], a 6-stage in-order RISC-V processor.

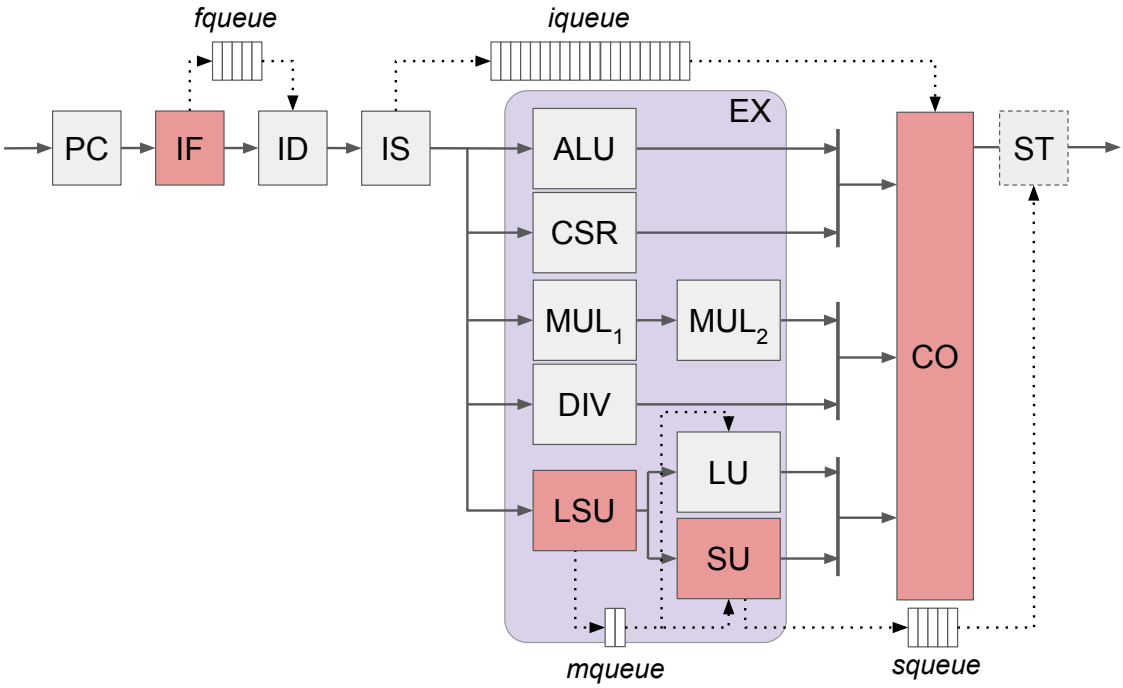


Figure 2.1: Model of the Ariane$^+$ core pipeline.

2.1.3.1 The original Ariane architecture

The structure of the Ariane core is depicted in Figure 2.1. The address of the next instruction to be fetched is computed in the first stage (PC). The instruction fetch (IF) stage hosts a branch predictor composed of a branch history table (BHT), a branch target buffer (BTB), a return address stack (RAS), and a static predictor (forward branches are predicted not taken, backward branches are predicted taken) which is used if the counter in the BHT has never been updated. The BHT and the BTB are updated each time a branch is resolved by the branch unit (i.e. when it reaches the end of the execution stage). Fetched instructions enter a 4-slot instruction queue (*fqueue*) which they exit in the instruction decode (ID) stage.

An 8-slot scoreboard holds all decoded instructions until they are committed. The issue stage (IS) inserts instructions into the scoreboard and dispatches them to the appropriate functional unit (FU).

The execution stage consists of a load-store unit (LSU), an ALU, a multiplier/divider and a CSR unit (that executes the instructions that access Control/Status Registers). The last three units are seen as a single functional unit by the issue stage: the Fixed Latency Unit (FLU)[1]. The ALU executes instructions in one cycle. Conditional branches are handled by a branch unit that uses the ALU to perform comparisons. The multiplier/divider is composed of a 2-stage multiplier and a non-pipelined, variable latency (2 to 64 cycles) divider.

The LSU is in front of a load unit (LU) and a store unit (SU). All memory instructions spend at least one cycle in the queue (*mqueue* which can hold at most 2 instructions) of the LSU before being dispatched to the LU or the SU. The LU sends a request to the data cache as soon as it receives a valid instruction. When an instruction hits in the data cache, its request is served in the current cycle, and the LU stage is available for a new instruction in the next cycle. In case of a miss however, the

[1]Even though the divider has a variable latency.

request is forwarded to the memory. When the data comes back from the memory, the instruction is allowed to leave the LU stage, but the stage is unavailable to a new instruction for an additional cycle. The SU keeps instructions in a 4-slot store buffer, that we describe in more details in Section 2.3. Additionally, atomic operations are kept in a separate buffer (AMO) of size one.

This design allows executing multiple instructions in parallel with the following restrictions:

- they do not depend on each other,
- their functional units do not share the same bus to write their results to the scoreboard, which prevents conflicts by design. The LU and SU share a bus, and the rest of the FUs share another bus,
- the SU cannot accept any instruction as long as the AMO buffer is not empty.

An instruction is allowed to enter the IS stage only if it is guaranteed that its FU will be available in the next cycle.

When an instruction has completed its execution, it remains in the scoreboard until it is the oldest instruction there. It is then processed by the commit stage (CO): results are written back to the register file, accesses to the CSR register file are performed, and entries in the store buffer are allowed to be written to the memory. Until they are committed, the results of completed instructions are forwarded to the functional units if needed. In case of a Write-after-Write hazard between two instructions, the youngest instruction (the one that enters the pipeline last) is stalled before entering the IS stage until the oldest has been committed (and has exited the CO stage).

The baseline version of Ariane that we use implements the RV32IMAC instruction set [72]. It does not rename registers, has no MMU, no FPU, and has a single commit port.

Releasing constraints on the functional units bus: Ariane$^+$ As mentioned earlier in this section, the functional units composing the FLU do not have the same latency. To avoid collisions on their shared bus (to write their results to the scoreboard), the scoreboard prevents instructions from entering the IS stage if they may spend more than one cycle in it (because their functional unit is currently in use) or whenever there is a risk that they request the bus at the same time as a pending instruction in the FLU units. This causes a slight decrease in performance.

We reduced the performance impact of this design by allowing instructions to enter the IS stage as long as the scoreboard is not full. In order to prevent collisions, we added a new write port to the scoreboard as well as a new bus dedicated to the ALU and the CSR, thus allowing the ALU or the CSR and the MUL/DIV units to write their results in parallel if needed. This mechanism guarantees that instructions leaving their functional unit cannot be delayed by younger instructions. In the case of Write-after-Write hazards between two instructions, the youngest instruction is now delayed inside the IS stage until the oldest is committed. In the remainder of the chapter, we call Ariane$^+$ the version of the core that includes these modifications.

2.1.3.2 Memory bus conflicts in the Ariane$^+$ core

A source of timing anomalies for in-order cores is when an instruction (e.g. a *load* or a *store*) that needs to access the memory bus is delayed by a subsequent instruction (typically when the code of this instruction is fetched from the memory) [43]. We refer to this phenomenon as an *inversion*.

We illustrate how inversions can lead to timing anomalies in Figure 2.2. This figure displays the execution timing of a simple sequence composed of 7 instructions in the pipeline of Ariane$^+$. We added a retire (RE) stage in order to show that the instructions are retired in order. Instructions 1 and 4

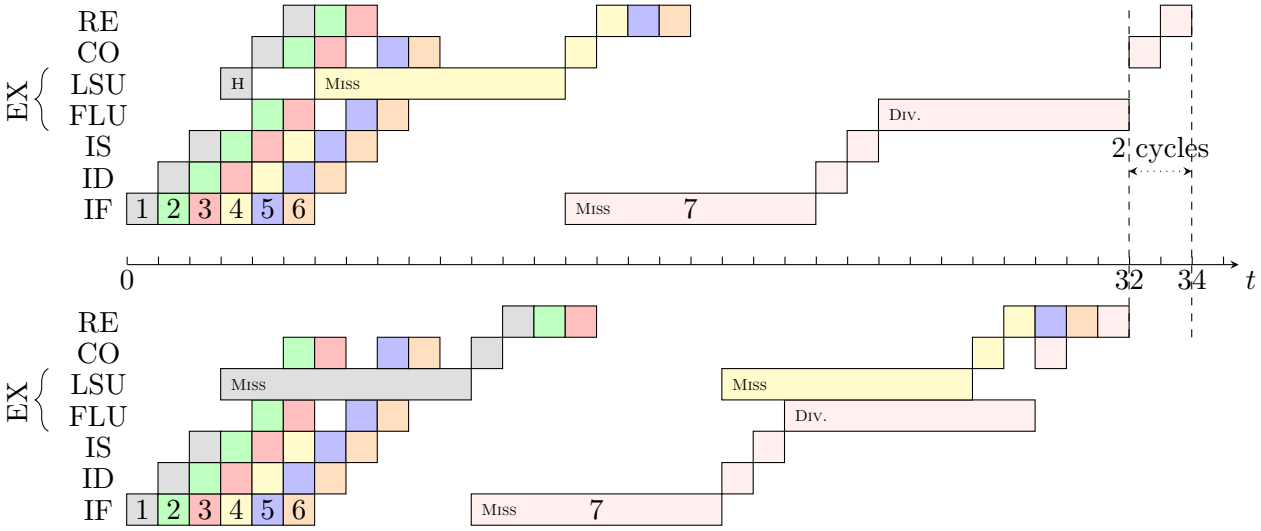


Figure 2.2: Example of a timing anomaly on the Ariane$^+$ core.

are memory loads, instruction 7 is a division. The rest of the instructions (2, 3, 5 and 6) are integer additions. We assume that instruction 7 leads to a cache miss in the FE stage and that instruction 4 leads to a cache miss in the LU stage.

At the top of Figure 2.2, we display the execution of the sequence if instruction 1 leads to a hit in the data cache (in the LU stage). Instruction 4 leads to a miss in the data cache at cycle 6. Then, instruction 7 must wait for instruction 4 to free the bus before it can enter the IF stage (because instruction 7 leads to a miss in the instruction cache). As a consequence, the sequence ends by instruction 7 being retired at cycle 34.

At the bottom of Figure 2.2, we display the execution timing of the sequence if instruction 1 leads to a miss in the data cache (in the LU stage). This miss blocks the fetch of instruction 7. When instruction 1 leaves the LU stage, instruction 4 cannot enter it because of the 1-cycle stall mentioned in section 2.1.3.1, so instruction 7 enters the IF stage and produces a miss in the instruction cache. This miss postpones the execution of instruction 4 in the LU stage: this is an inversion. This situation allows instruction 7 to enter the pipeline at date 11, and to later execute in parallel with the miss produced by instruction 4.

This example shows a timing anomaly in the Ariane$^+$ (also present in the original Ariane) core: a data cache miss for instruction 1 leads to an execution time of 32 cycles for the sequence, while a cache hit for the same instruction leads to an execution time of 34 cycles. Note that an inversion does not necessarily generate a timing anomaly in practice, but the fact that inversions happen makes it difficult to prove the absence of timing anomalies.

We added a new hardware counter (CSR) to the Ariane$^+$ processor to count for inversions and used the methodology described in Section 2.4.2.1. Over 52 TACLe benchmarks, 20 had inversions during their execution on the FPGA. This reveals that Ariane$^+$ is subject to timing anomalies and motivates our work to make it timing predictable.

2.1.4 Chapter organization

In the remainder of the chapter, we start (in Section 2.2) by introducing the MINOTAuR core, a modified version of Ariane$^+$ that enables timing predictability while allowing speculative execution. In the same section, we also present hardware mechanisms for LRU caches and RAS that allow their

use in MINOTAuR without breaking its predictability. We then study the effect of store buffers on the predictability of cores in Section 2.3. Finally, we present experimental results that measure the performance loss due to the gating mechanisms we introduced in MINOTAuR in Section 2.4.

2.2 Managing speculative execution in a predictable core

The MINOTAuR predictable processor is obtained from the Ariane$^+$ core by applying some restrictions to the pipeline. As stated earlier, the key idea to enforce timing predictability is to ensure that no instruction can be delayed by subsequent instructions. In Ariane$^+$, this amounts to suppressing inversions on the memory bus. To do so, we modified the IF stage so that it blocks instruction fetches when they are not already in the instruction cache and there is a pending memory instruction in the pipeline. Additionally, speculative execution is also blocked at the IF stage, unless the instruction is already in the instruction cache. This way, the instruction cache cannot send a request on the memory bus speculatively or when a memory instruction is already in the pipeline: we can guarantee the absence of inversions on the bus while tolerating a certain level of speculative execution. Another source of timing anomalies on the memory bus, store buffers, is discussed in lengths in Section 2.3.

This section is organized as follows: we start by providing the formal model of MINOTAuR in Section 2.2.1, and then prove its timing predictability in Section 2.2.2. Since these proofs rely on a simplifying hypothesis that limits its applicability, in Section 2.2.3 we explore new designs for the instruction cache and the RAS of the core that allow us to relax these hypotheses.

2.2.1 Formal model of MINOTAuR

2.2.1.1 Non-speculative components of the pipeline

Each instruction $i \in \mathcal{I}$ is characterized by its category $opc(i) \in \{alu, branch, store, load, atomic, mul, div, csr\}$ and by predicates that reflect the outcome of the cache analysis: $ichit(i)$ (resp. $dchit(i)$) is true if the cache analysis has determined that instruction i resides in the instruction cache (resp. the data accessed by instruction i resides in the data cache).

The complete formal model of the MINOTAuR core is shown in Figure 2.3. This model specifies the pipeline structure[2] and the *cycle* function with the help of the following auxiliary predicates and functions that are defined for a given pipeline state $c \in \mathcal{C}$:

- $c.isnext(i, s)$: true if instruction i is the oldest in stage s
- $c.nstg(i)$: next pipeline stage for instruction i. It depends on its current stage and sometimes on its category.
- $c.cnt(i)$: number of cycles that instruction i still has to spend in the stage it currently resides in.
- $c.nlat(i)$: latency of instruction i in its next pipeline stage. Only memory instructions and divisions have a non-zero latency in their functional unit. The latency of an instruction fetch is determined by the latency to the main memory in case of a cache miss.
- $c.pending(i, op)$: true if an instruction of category op and older than i has not been completely processed in a given stage defined by $lstg(op)$. $lstg(op)$ maps each category of instruction op to the last stage before committing such an instruction. *Stores* and *atomic* instructions are pending

[2]The *pre* (resp. *post*) stage hosts instructions that have not yet entered (resp. have left) the pipeline.

until they have been sent to the memory (in stage ST). Instructions accessing hardware counters (*csr*) are pending until they are committed. All other instructions are pending until they have been processed by their functional units.

- *c.ready*(i): true if instruction i is ready to advance to the next pipeline stage. For most of the pipeline stages, an instruction is ready when it has been completely processed by the stage and when it is the oldest one in the stage (this condition is required for stages that host several instructions). In addition, there are restrictions to advance from PC to IF (no pending branch, and if the instruction misses in the cache, no pending memory instruction), from IS to the functional units (multiplications and divisions are stalled if there is a pending division currently in the division unit, and instructions are stalled if a dependency exists with a previous instruction – modelled by the $dep_{RaW}(i, j)$ (resp. $dep_{WaW}(i, j)$) predicate – that has not completed its execution (resp. exited the CO) stage yet), and from LSU to LU or SU (*loads* are stalled by pending *stores*, and *loads* and *stores* are stalled by pending *atomic* instructions).

- *c.slot*(s): for any pipeline stage s that inserts instructions in a queue/buffer, true when the queue/buffer will have a free slot in the next clock cycle. This is determined by counting the number of instructions that reside between the entering and leaving pipeline stages and by checking whether an instruction that is already in the queue will leave it and release a slot. The size of the *fqueue* (resp. *mqueue*, *iqueue*, *squeue*) is denoted fq_size (resp. mq_size, iq_size, sq_size) in the model.

- *c.free*(s): true if stage s can accept a new instruction in the next clock cycle. Some of the stages always accept instructions, either because they can host several of them or because they keep instructions for a single cycle. Other stages insert instructions in a queue, and it must be guaranteed that this queue has a free slot. Finally, for other stages, one checks whether the instruction they currently host will be able to advance to its next stage. For the LU stage, the 1-cycle stall after a miss is also modelled.

The MINOTAuR core features several instructions queues that improve its throughput. We model them by considering that an instruction that resides in a queue stays in a given pipeline stage when it is not currently processed. For example, fetched instructions are inserted in the *fqueue* in stage IF and remain there until they enter the ID stage. The scoreboard is represented by the *iqueue* which instructions enter in IS and leave in stage CO. Similarly, memory instructions enter the *mqueue* in stage LSU and leave it when they advance to the LU/SU unit. The store buffer is modeled as an instruction queue, *squeue*, and a fictive store stage (ST) that represents the actual sending of write requests to the memory. All this means that we allow several instructions to reside in the same stage, even if only the youngest one is effectively processed by the stage. We keep track of the number of instructions in each stage using set cardinals (#). Pipeline stages that can host several instructions (one being effectively processed and the other being only hosted) are shown in light red in Figure 2.1.

$$\mathcal{S} := \{pre, \text{PC}, \text{IF}, \text{ID}, \text{IS}, \text{ALU}, \text{MUL}_1, \text{MUL}_2, \text{DIV}, \text{LSU}, \text{LU}, \text{SU}, \text{CSR}, \text{CO}, \text{ST}, post\}$$

$$pre \sqsubset_{\mathcal{S}} \text{PC} \sqsubset_{\mathcal{S}} \text{IF} \sqsubset_{\mathcal{S}} \text{ID} \sqsubset_{\mathcal{S}} \text{IS} \sqsubset_{\mathcal{S}} \{\text{ALU}, \text{MUL}_1, \text{LSU}, \text{CSR}, \text{DIV}\} \sqsubset_{\mathcal{S}} \{\text{MUL}_2, \text{LU}, \text{SU}\} \sqsubset_{\mathcal{S}} \text{CO} \sqsubset_{\mathcal{S}} \text{ST} \sqsubset_{\mathcal{S}} post$$

$$cycle(c)(i) := \begin{cases} (c.nstg(i), c.nlat(i)) & : c.ready(i) \wedge c.free(c.nstg(i)) \\ (c.stg(i), c.ncnt(i)) & : otherwise \end{cases}$$

$$c.isnext(s,i) := c.stg(i) = s \wedge \forall j < i \,.\, c.stg(j) \sqsupset_{\mathcal{S}} s$$

$$c.ncnt(i) := \begin{cases} c.cnt(i) - 1 & : c.cnt(i) > 0 \\ 0 & : otherwise \end{cases}$$

$$c.nlat(i) := \begin{cases} memlat_f(i) & : c.nstg(i) = \text{IF} \wedge \neg ichit(i) \\ memlat_d(i) & : (c.nstg(i) = \text{LU} \wedge \neg dchit(i)) \\ & \quad \vee c.nstg(i) = \text{ST} \\ exlat(i) & : c.nstg(i) = \text{DIV} \\ 0 & : otherwise \end{cases}$$

$$c.pending(i, op) := \exists j < i \,.\, opc(j) = op \wedge c(j) \sqsubset_{\mathcal{P}} (lstg(op), 0)$$

$$c.nstg(i) := \begin{cases} post & : c.stg(i) \neq pre \wedge \neg c.pending(i, branch) \wedge pwrong(i) \\ c.nstg'(i) & : otherwise \end{cases}$$

$$c.nstg'(i) := \begin{cases} \text{PC} & : c.stg(i) = pre \\ \text{IF} & : c.stg(i) = \text{PC} \\ \text{ID} & : c.stg(i) = \text{IF} \\ \text{IS} & : c.stg(i) = \text{ID} \\ \text{LSU} & : c.stg(i) = \text{IS} \wedge opc(i) \in \{load, store, atomic\} \\ \text{LU} & : c.stg(i) = \text{LSU} \wedge opc(i) = load \\ \text{SU} & : c.stg(i) = \text{LSU} \wedge opc(i) \in \{store, atomic\} \\ \text{MUL}_1 & : c.stg(i) = \text{IS} \wedge opc(i) = mul \\ \text{MUL}_2 & : c.stg(i) = \text{MUL}_1 \\ \text{DIV} & : c.stg(i) = \text{IS} \wedge opc(i) = div \\ \text{CSR} & : c.stg(i) = \text{IS} \wedge opc(i) = csr \\ \text{ALU} & : c.stg(i) = \text{IS} \wedge opc(i) \notin \{load, store, atomic, mul, div, csr\} \\ \text{CO} & : c.stg(i) \in \{\text{ALU}, \text{MUL}_2, \text{DIV}, \text{CSR}, \text{LU}, \text{SU}\} \\ \text{ST} & : c.stg(i) = \text{CO} \wedge opc(i) \in \{store, atomic\} \\ post & : (c.stg(i) = \text{CO} \wedge opc(i) \notin \{store, atomic\}) \vee (c.stg(i) = \text{ST}) \end{cases}$$

$$lstg(op) := \begin{cases} \text{LU} & : op = load \\ \text{ST} & : op = store \\ \text{ST} & : op = atomic \\ \text{MUL}_2 & : op = mul \\ \text{DIV} & : op = div \\ \text{CO} & : op = csr \\ \text{ALU} & : op = branch \\ \text{ALU} & : op = alu \end{cases}$$

$$\begin{aligned} c.ready(i) := {} & (c.stg(i) \neq pre \wedge \neg c.pending(i, branch) \wedge pwrong(i)) \\ & \vee (c.cnt(i) = 0 \wedge c.isnext(c.stg(i), i)) \\ & \quad \wedge (c.stg(i) = \text{PC} \Rightarrow (ichit(i) \\ & \qquad \vee (\neg c.pending(i, branch) \wedge \neg c.pending(i, load) \wedge \neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \\ & \quad \wedge (c.stg(i) = \text{IS} \Rightarrow (opc(i) \in \{mul, div\} \Rightarrow \neg c.pending(i, div)) \\ & \qquad \wedge \forall j < i \,.\, ((dep_{WaW}(i,\, j) \Rightarrow c.stg(j) \sqsupset_{\mathcal{S}} \text{CO}) \\ & \qquad\quad \wedge (dep_{RaW}(i,\, j) \Rightarrow ((opc(j) = csr \wedge c.stg(j) \sqsupset_{\mathcal{S}} \text{CO}) \vee (c.stg(j) \sqsupseteq_{\mathcal{S}} \text{CO})))) \\ & \quad \wedge (c.stg(i) = \text{LSU} \Rightarrow (opc(i) \in \{store, atomic\} \wedge \neg c.pending(i, atomic)) \\ & \qquad \vee (opc(i) = load \wedge (\neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \end{aligned}$$

$$\begin{aligned} c.free(s) := {} & s \in \{\text{ALU}, \text{MUL}_1, \text{CSR}, \text{MUL}_2, \text{CO}, post\} \\ & \vee (s \in \{\text{IF}, \text{IS}, \text{LSU}, \text{SU}\} \wedge c.slot(s)) \\ & \vee (s \in \{\text{PC}, \text{ID}, \text{DIV}, \text{ST}\} \wedge ((\neg\exists j \,.\, c.stg(j) = s) \vee (\exists j \,.\, c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j))))) \\ & \vee (s = \text{LU} \wedge ((\neg\exists j \,.\, c.stg(j) = \text{LU}) \vee (\exists j \,.\, c.stg(j) = \text{LU} \wedge c.ready(j) \wedge c.free(c.nstg(j)) \wedge dchit(j)))) \\ & \vee (\exists i. c.stg(i) = s \wedge pwrong(i) \wedge \neg c.pending(i, branch)) \end{aligned}$$

$$\begin{aligned} & c.slot(\text{IF}) := ((\#\{j | c.stg(j) = \text{IF}\} < fq\_size) \vee c.free(\text{ID})) \wedge \forall j. c.stg(j) = \text{IF} \Rightarrow c.cnt(j) = 0 \\ & c.slot(\text{IS}) := \#\{j | \text{IS} \sqsubseteq_{\mathcal{S}} c.stg(j) \sqsubseteq_{\mathcal{S}} \text{CO}\} < iq\_size \vee (\exists j' \,.\, c.isnext(\text{CO}, j') \wedge c.ready(j') \wedge (opc(j') \in \{store, atomic\} \Rightarrow c.free(\text{ST}))) \\ & c.slot(\text{SU}) := \#\{j | opc(j) = store \wedge \text{LSU} \sqsubset_{\mathcal{S}} c.stg(j) \sqsubset_{\mathcal{S}} post\} < sq\_size \vee \exists j' \,.\, c(j') = (\text{ST}, 0)) \\ & c.slot(\text{LSU}) := \#\{j | c.stg(j) = \text{LSU}\} < mq\_size \\ & \qquad \vee (\exists j' \,.\, c.isnext(\text{LSU}, j') \wedge ((opc(j') = load \wedge c.free(\text{LU})) \vee (opc(j') \in \{store, atomic\} \wedge c.free(\text{SU})))) \end{aligned}$$

Figure 2.3: Model of the MINOTAuR core.

2.2.1.2 Timing predictable speculative execution

As pointed out in Section 2.1.3, MINOTAuR features a branch predictor that is the support for speculative execution. We say that an instruction is *speculated* if the pipeline contains an older, still unresolved branch. We say that the instruction is *misspeculated* if the unresolved branch has been mispredicted, i.e. if the instruction belongs to the wrong path. In order to deal with mispredictions, we introduce a new predicate, $pwrong(i)$ that works in the same way as $ichit(i)$ and $dchit(i)$. The predicate $pwrong(i)$ is true whenever instruction i is misspeculated. Using this predicate, any misspeculated instruction that has already entered the pipeline is directly flushed to the *post* stage (i.e. exits the pipeline without being executed or committed) as soon as the branch has been resolved. In the *ready* function, an instruction i is allowed to enter the IF stage even speculatively as long as $ichit(i)$ is true. On the contrary, if the instruction is going to cause a miss in the instruction cache, it is stalled in the PC stage as long as a *branch* or a memory (*load*, *store*, *atomic*) instruction is pending. The portions of the model that relate to these aspects of speculative execution are highlighted in blue in Figure 2.3.

Allowing some instructions to enter the pipeline speculatively does not affect the timing predictability of the core as long as these speculated instructions do not modify the state of the hardware (except for the pipeline contents). In that regard, the RAS incurs a difficulty: it is updated in the early stages of the pipeline, before knowing if the corresponding function call itself is executed as part of a mispredicted branch. Moreover, the effect of speculated instructions on the instruction cache contents and inner state (e.g. blocks ages) must be considered. As MINOTAuR lets instructions enter the IF stage speculatively only when they result in a hit in the instruction cache, its contents are not modified during speculative execution. However, if the cache features an aging mechanism (e.g. an LRU cache), its state may be modified by a hit during the speculation.

In the next section, we prove the timing predictability of the MINOTAuR core, assuming two important restrictions: (i) that the RAS is disabled, and (ii) that the effect of cache hits on the instruction cache state is transparent to usual cache analysis [53] i.e. cache hits do not affect the cache state in a way that is not modeled by the analysis (e.g. direct-mapped or random caches such as the ones implemented in Ariane). We then describe and evaluate general mechanisms that can be added to any RAS or cache in order to lift these restrictions.

As the design and effects of the store buffers can be quite complex, we dedicate Section 2.3 of this manuscript to the modelling and modification of store buffers regardless of the presence of speculation. Note however that in the model of Figure 2.3 we already included the gating mechanism for the store buffer that we present in detail in Section 2.3, so that the model is complete and the timing predictability of the core can be proven. Additionally, as speculated store instructions cannot perform their write to memory (in stage ST, i.e. after stage CO) before the corresponding branch instruction is resolved, we do not need to consider the effect of speculated stores in these proofs.

2.2.2 Timing anomaly freedom of MINOTAuR

We list here a series of theorems used to prove the timing predictability of MINOTAuR. The proofs for these theorems were published in the IEEE Transactions on Computers journal [34].

The first theorem states that caches cannot be modified by speculated instructions.

Let $c \in \mathcal{C}$ be a pipeline state and $i \in \mathcal{I}$ be an instruction. The state of the instruction or data

cache might be modified by i if and only if the following predicate is true:

$$c.cmod(i) := (c.stg(i) = \text{IF} \wedge \neg ichit(i)) \vee (c.stg(i) = \text{LU} \wedge \neg dchit(i))$$

Theorem 6: Absence of cache state modification during speculation

$\forall i \in \mathcal{I}, \forall c \in \mathcal{C}, c.pending(i, branch) \Rightarrow \neg cycle(c).cmod(i)$

It results from this theorem and its proof that (i) no request to the memory can be initiated by a speculated instruction and thus no memory request started speculatively is pending at the time when the corresponding branch is resolved, (ii) speculated instructions are not subject to multi-core interference and (iii) uncertain outcomes of the cache analyses can be treated as part of the non-speculative execution.

The next theorem states that MINOTAuR follows the Update enable property.

Theorem 7: Update enable in MINOTAuR

The MINOTAuR core satisfies Property 1.

Using Theorem 7, we have that the MINOTAuR core satisfies Property 1, and using Theorem 6 that we do not have to consider the hypothetical case of non-determinism in the caches or memory latencies for speculated instructions.

Next, we prove that allowing speculation as specified in the model does not introduce timing anomalies in the core. To do this, we consider an instruction sequence $\mathcal{I}_1 := i_1, i_2, ..., i_{br}, i_{br+1}, ..., i_n$ in which i_{br} is the only branch instruction, and we make the assumption that the prediction on this branch can be either correct or incorrect. $\mathcal{I}_1$ itself represents the execution when the prediction is correct. A second sequence $\mathcal{I}_2 := i_1, i_2, ..., i_{br}, m_1, m_2, ..., m_k, i_{br+1}, ..., i_n$ contains misspeculated instructions (m_x) that may enter the pipeline if the prediction is wrong. We denote c_{br} the state of the pipeline when i_{br} enters the IF stage. It is important to remark that all instructions $i \leq i_{br}$ are identical in both sequences, and that the same is true for instructions $i \geq i_{br+1}$.

Let c_w be the state of the pipeline just when i_{br} has been resolved ($c_w(i_{br}) = (\text{ALU}, 0)$) if it has been mispredicted (i.e. the local worst case). Without loss of generality, we assume that c_w is obtained by applying the *cycle* function $l > 0$ times on c_{br} while following the $\mathcal{I}_2$ sequence. Additionally, let c_b be the state of the pipeline just when i_{br} has been resolved ($c_b(i_{br}) = (\text{ALU}, 0)$) if it has been predicted correctly (i.e. the best local case). Since all instructions $j < i_{br}$ are the same in $\mathcal{I}_1$ and $\mathcal{I}_2$ and the pipeline implements the progress dependence property, c_b is also obtained by applying the *cycle* function l times on c_{br}, but this time following the $\mathcal{I}_1$ sequence. Since both sequences are identical up to i_{br}, these two states correspond to the same number of applications of *cycle* since the beginning of the execution. By considering c_w and c_b, we can prove progress properties without having to consider the speculated instructions: we compare c_w and c_b only on the instructions that they have in common i.e. the instructions of $\mathcal{I}_1$.

Theorem 8: Progress at the end of speculation

Pipeline state c_w has less progress on $\mathcal{I}_1$ than c_b : $c_w \sqsubseteq c_b$. More precisely:

$$\forall j \in \mathcal{I}_1, \begin{cases} j \leq i_{br} \Rightarrow c_w(j) = c_b(j) \\ j > i_{br} \Rightarrow c_w(j) \sqsubseteq_{\mathcal{P}} c_b(j) \end{cases}$$

Theorem 6 guarantees that caches are not modified during speculation, and we know that by design the dynamic branch prediction mechanisms are only updated when branches are resolved, with the information of the correct branch. This means that any modification of these components that could impact the execution of subsequent instructions (e.g. cache content modification) cannot happen during speculation. Using Theorem 8, we can thus safely apply function f of Theorem 2 to c_b and c_w and conclude on the absence of timing anomalies in MINOTAuR.

We now adapt and prove Theorem 5 for MINOTAuR.

Theorem 9: Compositionality of MINOTAuR w.r.t. cache uncertainty

Let two valuations of dchit (or ichit) be given that differ for an arbitrary instruction $i \in \mathcal{I}$. The valuation that predicts a cache miss will lead to a finishing time at most p cycles higher than the valuation that predicts a cache hit.

We finally proceed with the last theorem that bounds the timing penalty for a branch misprediction in MINOTAuR.

Theorem 10: Bound of the timing penalty resulting from a branch misprediction

If a predicted branch takes p cycles to be resolved, then the penalty for a misprediction of the branch is at most p cycles.

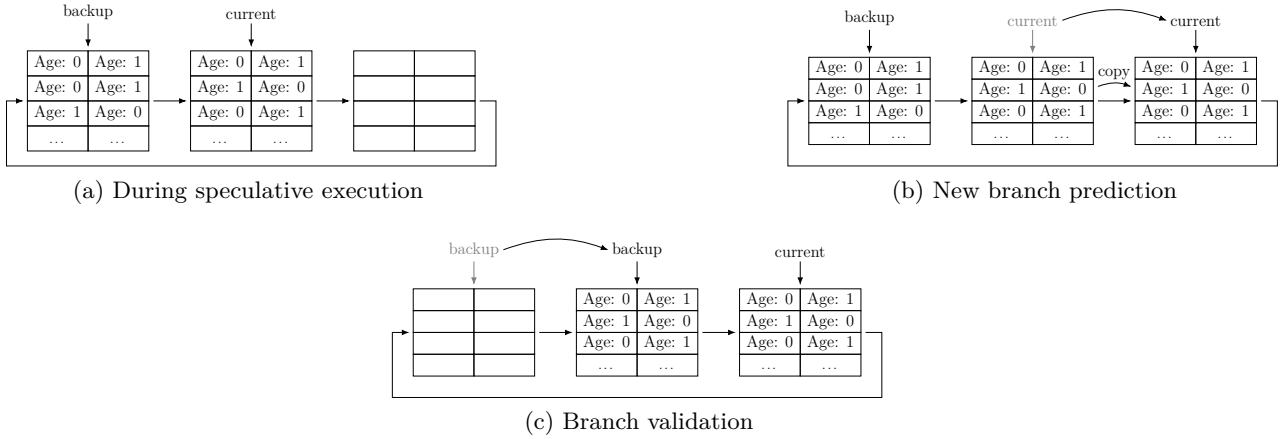
2.2.3 Releasing the constraints on the RAS and caches

In Section 2.2.1.2, we made the assumption that the RAS was disabled and that the caches did not implement aging mechanisms. We now present hardware mechanisms that allow lifting these restrictions while keeping MINOTAuR timing predictable.

2.2.3.1 Speculation-aware cache state backups

A cache hit may modify the state of caches implementing an aging-based replacement policy (such as LRU). This is problematic as MINOTAuR does not stall speculated instructions that hit in the instruction cache. As a result, a misspeculated instruction could modify the age of blocks in the instruction cache. Then, when the corresponding branch is resolved and the core starts executing on the correct path, the blocks ages would be different from what they were before the speculation began. This can have an impact on the selection of the next evicted blocks, and ultimately on the timing of the instruction sequence. Consequently, our proofs no longer hold in this context.

To solve this issue, we designed a hardware mechanism that makes a backup copy of the ages of the cache blocks each time a branch prediction is made. When a branch is resolved, the backup is restored if the prediction was incorrect. Otherwise, the current state of the cache is committed and the backup invalidated. In order to support nested branches, the backup mechanism is implemented using a circular buffer of copies. We illustrate the behavior of the backup mechanism in Figure 2.4. In this example, we consider the cache state backup mechanism evolution while a sequence of instructions is executed. We assume a 2-way set associative cache featuring an aging-based replacement policy. Since cache misses are blocked during speculative execution, the backup mechanism only needs to save the ages of the blocks (and not their contents). The circular buffer implementing the backup mechanism in this example can hold up to 3 copies of the cache state. In Figure 2.4 (a), the core is currently executing an instruction speculatively. The backup mechanism keeps a safe copy of the ages of the blocks when the speculation started. This copy is identified using a “backup” pointer.



(a) During speculative execution

(b) New branch prediction

(c) Branch validation

Figure 2.4: Cache state backup mechanism.

All modifications to the ages done speculatively are accounted for in another copy designated by the "current" pointer. In the example, four blocks have their age modified in the current copy, compared to when the speculation started. If a new branch instruction is executed before the previous one has been resolved, the contents of the current copy is duplicated to another copy, and the "current" pointer is incremented to point to that new copy (Figure 2.4 (b)). Now, when the first branch is resolved, if the prediction was correct, the "backup" pointer is incremented to the next copy in the buffer (Figure 2.4 (c)). If however the prediction was incorrect, the "backup" pointer does not move, an the "current" pointer is set to the copy pointed by the "backup" pointer.

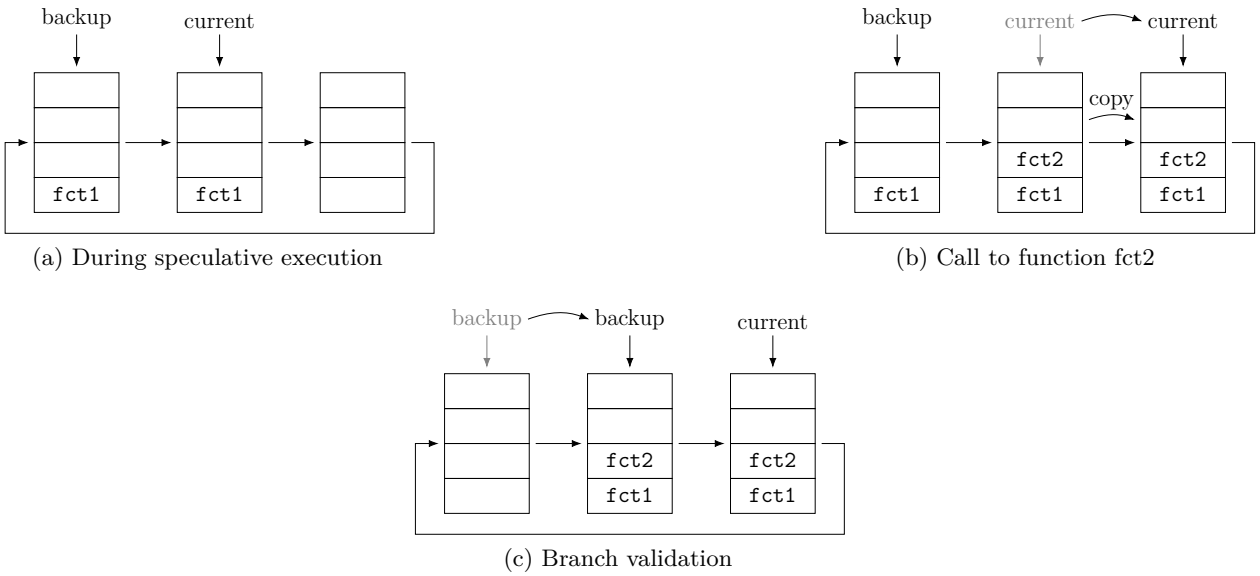
When the buffer that holds the copies is full (as the result of too many nested branches), new branch instructions are blocked before they can enter the IF stage to prevent any unsaved modification to the cache state. When a pending branch is resolved, the buffer recovers at least one slot, and branch instructions are allowed to enter the IF stage again.

The backup mechanism is designed to work with any cache that implements an aging-based replacement policy (e.g. pseudo-LRU, Most Recently Used). We implemented and tested it on a LRU cache, because this policy is particularly fitted for static WCET analysis. The results of our evaluation are given in Section 2.4.

2.2.3.2 RAS backup mechanism

A return address stack (RAS) is a branch prediction mechanism used to predict the return address at the end of a function call: when a return instruction (e.g. `jr ra` in RISC-V) is executed at the end of a function, the RAS predicts the address of the next instruction to execute. In a simple RAS, as found in the Ariane processor, the address of the next instruction is pushed onto the stack when a function call (e.g. `jal`, `jalr` in RISC-V) enters the fetch stage. The return address is popped from the RAS when a return instruction is fetched, and the next PC is set to this address.

Problems can come from the speculative execution of branches. Recall that the RAS is updated when function call or return instructions enter the first stages of the pipeline. If these instructions are misspeculated (i.e. they should not be executed), the RAS gets updated with incorrect information. The first problematic situation happens when a return instruction is misspeculated: in this case, the return address at the top of the RAS is popped. When the corresponding branch is resolved this popped entry is not restored on the stack: the RAS has lost information. When the correct branch is executed and the return instruction enters the frontend, the top of the stack does not contain

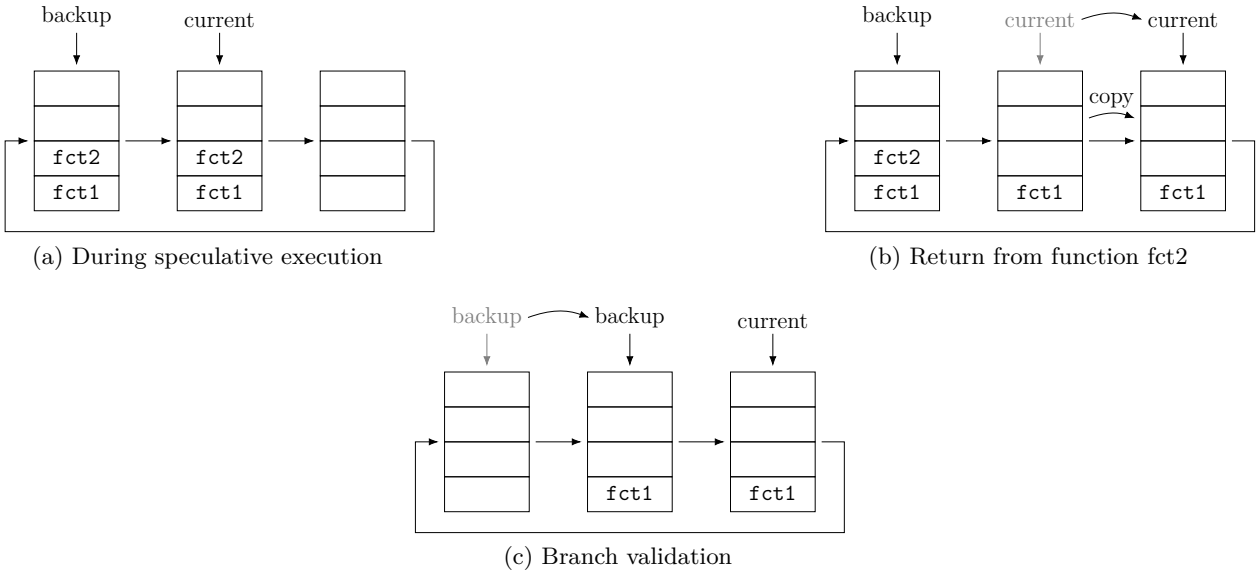


(a) During speculative execution

(b) Call to function fct2

(c) Branch validation

Figure 2.5: RAS state backup mechanism: function call.



(a) During speculative execution

(b) Return from function fct2

(c) Branch validation

Figure 2.6: RAS state backup mechanism: return from function.

the corresponding return address. The second problematic situation happens when a function call is misspeculated. In this case, the corresponding return address is pushed to the stack. When the speculation ends and the correct branch starts executing, this entry remains in the stack: the RAS contains a return address that corresponds to no function call.

To avoid these situations, we implement a backup mechanism similar to the one we described for the LRU caches, but with additional subtleties. Function calls and returns are implemented as branch instructions and may be speculated in the case of indirect calls (e.g. function pointers). In both the cache and the RAS backups, the "current" pointer is incremented each time a prediction is made, including when function calls and function returns are fetched. However, when an indirect

function call is made, the contents of the RAS are updated when the corresponding branch instruction is fetched. If an incorrect prediction is made for the branch target, the branch instruction is not fetched again when the control is set to the correct address, and the RAS is not updated at this point. As a consequence, our backup mechanism updates the RAS with the return address of a function call before the "current" pointer is incremented, and the RAS is copied to a new backup slot. This way the return address is present in two backup copies. Whether the prediction is correct or not, the RAS is in the correct state when the corresponding return instruction enters the pipeline. One important point here is that regardless of the target of the branch for a function call, the return address is the same, so we can safely push it on the RAS. This is illustrated in Figure 2.5. To remain coherent, our backup mechanism handles return instructions in the same way. When a return instruction enters the pipeline, the top element from the "current" copy of the RAS is popped. Then, since the return instruction is a branch, the "current" pointer is incremented and the contents of the RAS are copied. This is illustrated in Figure 2.6.

Now that we have proven the timing predictability of MINOTAuR, including during speculative execution, and that we have proposed hardware designs that enable the predictable use of RAS and LRU caches in the core, we are going to take a closer look at a component that we have eluded so far, and at its effect on timing predictability: the store buffer.

2.3 Managing store buffers in a predictable core

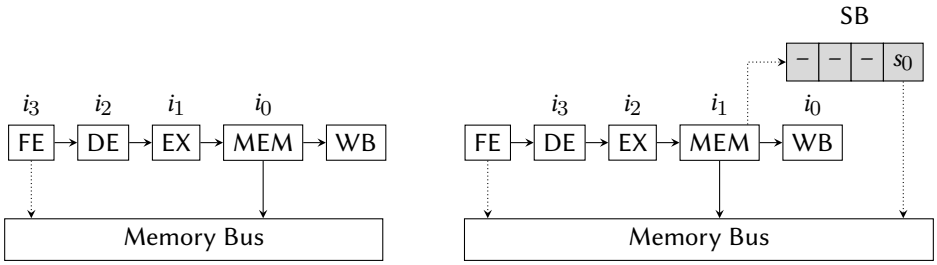
In a processor pipeline, a store buffer [10] allows store instructions to leave the memory stage and be committed while the write request has not been sent to the memory yet. This way, the write latency is hidden. However, since store buffers are connected to the memory bus in order to emit their write requests, they share this sequential resource with the instruction and data caches, and thus may create inversions that lead to timing anomalies. Since the problem between the instruction cache and all memory instructions (loads and stores) has been tackled in the previous chapter, we now focus on the concurrency between load and store instructions and their effect on the timing predictability of pipelines. In order to keep our work generic, we encompass both in-order pipelines and pipelines in which the load and store instructions are processed by separate, parallel functional units.

This section is organized as follows. In Section 2.3.1, we provide background information about store buffers, and the two pipeline settings that we are going to study. Then, in Section 2.3.2, we provide examples showing how store buffers break the progress monotonicity of pipelines, thus making them vulnerable to timing anomalies (or at least making it harder to prove their absence). In Section 2.3.3, we provide a gating mechanism that restores progress monotonicity in the presence of store buffers. Finally, in Section 2.3.4, we present the subtleties of the implementation of store buffers in MINOTAuR and make a formal link between the proposed gating mechanism and the model of the core provided in Figure 2.3 of the last section.

2.3.1 Generalities about store buffers

2.3.1.1 In-order pipelines

In a simple scalar in-order pipeline such as the textbook 5-stage pipeline considered in [46], all memory accesses (loads and stores) are processed in the same memory stage. Without a store buffer, a store to a memory block that does not reside in the data cache must stay in the memory stage for at least the latency of the access to the memory, stalling all subsequent instructions. This is illustrated in Figure 2.7(a): instruction i_0 is a store that accesses the memory bus and stalls the pipeline until the



(a) Without store buffer. (b) With a store buffer.

Figure 2.7: In-order pipeline

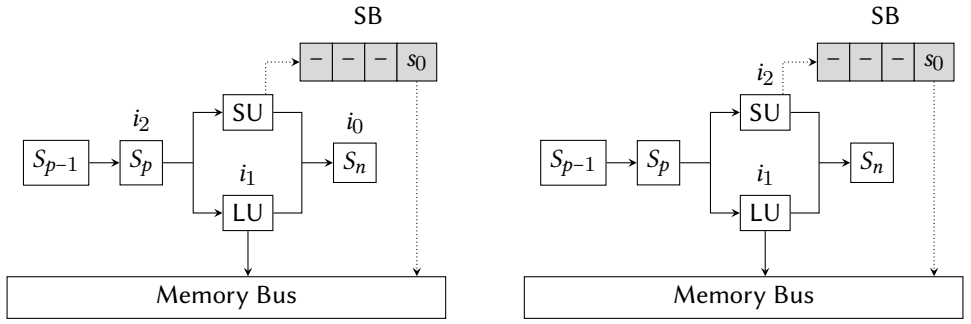
write to the memory is completed. Although this stall is usually necessary for loads (since the loaded data is likely to be used by the following instructions), stores can often be delayed safely.

A store buffer keeps track of the stores that have been executed in the pipeline but have not yet been sent to the memory. As a consequence, a store instruction to an uncached block does not wait in the memory stage until the block has been loaded or the data has been written to the memory: the address and data to be written are pushed to the store buffer, and the instruction goes to the next stage in the next clock cycle, allowing the flow of instructions to progress in the pipeline. The store buffer then sends the pending writes to memory whenever it gets access to the memory bus. This is illustrated in Figure 2.7(b): store instruction i_0 is not stalled in the MEM stage. Instead the write (s_0) is enqueued in the store buffer, and i_0 progresses to the WB stage. In the example, instruction i_1 enters the MEM stage and performs a load, thus requesting the memory bus. When the load is completed, i_1 advances to the WB stage and i_2 enters the MEM stage. If i_2 does not request the bus, the store buffer is able to send s_0 to the memory.

2.3.1.2 Out-of-order loads and stores

In more complex architectures, load and store instructions to different memory blocks may be executed out of order by using separate instruction queues for loads and stores, or a reorder buffer in the load/store unit (LSU). In this setup, a store buffer allows store instructions to advance in the pipeline even though an older load instruction may be using the memory bus. In order to avoid breaking memory dependencies, addresses of loads are checked against those of the stores in the store buffer. Figure 2.8 depicts an example of such a design, in which we represent separately the load unit (LU) and the store unit (SU). In order to remain as general as possible, we only depict the portion of the pipeline related to memory accesses. This portion may be inserted in a much longer pipeline and/or in parallel with other functional units. In Figure 2.8(a), instruction i_0 just advanced to stage S_n after enqueuing its store s_0 in the store buffer in the last cycle. Instruction i_1 (a load) just entered the LU stage and started using the bus, while instruction i_2 (another store) just entered stage S_p. The state of the pipeline in the next cycle is depicted in Figure 2.8(b): instruction i_0 advances past the S_n stage, while i_1 is still accessing the memory. i_2 is allowed to progress to the SU stage. In the following cycle, i_2 will enqueue its store in the store buffer. If i_1 still has not finished its access to the memory, i_2 will advance to stage S_n.

A consequence of this design is the possibility for a load to be delayed or not before accessing the bus, depending on the state of the store buffer. As we will see in Section 2.3.2, this feature is a problem for timing predictability. Additionally, depending on the design, a store instruction may be blocked in the LSU because the store buffer is full while a subsequent load may be allowed to use



(a) i_0 has enqueued its store in the store buffer and advanced to the S_n stage. In the meantime i_1 has advanced to the LU stage and started accessing the bus.

(b) i_0 advances past S_n. i_1 is still accessing the bus and i_2 progresses to the SU stage.

Figure 2.8: Out-of-order memory accesses

the memory bus, thus delaying the advance of the store instruction. This can also be problematic for timing predictability.

2.3.2 Store buffers effect on monotonicity

In this section, we provide formal models to characterize the effect of store buffers on the two architectural designs discussed in Section 2.3.1. In order to keep our results general, we do not provide the complete *cycle* function for a specific pipeline. Instead we focus on the portion of pipeline that is related to the store buffer, and make no particular assumption about the topology of the rest of the pipeline.

In order to model the occupancy of the memory bus (either by the store buffer or by the memory/load unit) in each execution cycle, we introduce the *busTaken* predicate. Given a pipeline state c_a and an instruction i such that $opc(i) = load$, $c_a.busTaken(i)$ is true if and only if i is using the memory bus in the current cycle. Note that for our proofs, we do not need a formula to compute the value of *busTaken*.

2.3.2.1 Simple in-order pipeline

We define the *cycle* function for the simplified 5-stage pipeline in Figure 2.9. As pointed out earlier, we only focus on the EX, MEM and WB stages. The previous stages are modeled by an abstract stage called *pre* that initially holds all instructions. The retired instructions go to an abstract stage called *post* after the WB stage. An instruction advances to the next stage when (1) it is ready to advance, and (2) the next stage is guaranteed to be free in the next cycle. The $ready(i)$ function first guarantees that instruction i has been processed in its current stage ($cnt(i) = 0$). Then, depending on the current stage of i, it checks:

- if i is in *pre*, that i is the oldest instruction in *pre* (instructions enter the pipeline in program order);
- if i is in MEM, that if i is a store, the store buffer is not full. This is done using the c.sbFull() predicate that evaluates to false iff the store buffer is currently not full, or if it is full but will no longer be in the next cycle.

$$\mathcal{S} := \{pre,\ EX,\ MEM,\ WB,\ post\}$$

$$pre \sqsubset_{\mathcal{S}} EX \sqsubset_{\mathcal{S}} MEM \sqsubset_{\mathcal{S}} WB \sqsubset_{\mathcal{S}} post$$

$$cycle(c)(i) := \begin{cases} (c.nstg(i),\ c.nlat(i)) & : c.ready(i) \wedge c.free(c.nstg(i)) \\ (c.stg(i),\ c.ncnt(i)) & : otherwise \end{cases}$$

$$c.ncnt(i) := \begin{cases} c.cnt(i) - 1 \text{: } c.cnt(i) > 0 \\ \qquad\qquad \wedge opc(i) = load \Rightarrow (c.stg(i) \neq MEM \vee c.busTaken(i) \vee dchit(i)) \\ c.cnt(i) & : opc(i) = load \wedge c.stg(i) = MEM \wedge \neg c.busTaken(i) \wedge \neg dchit(i) \\ 0 & : otherwise \end{cases}$$

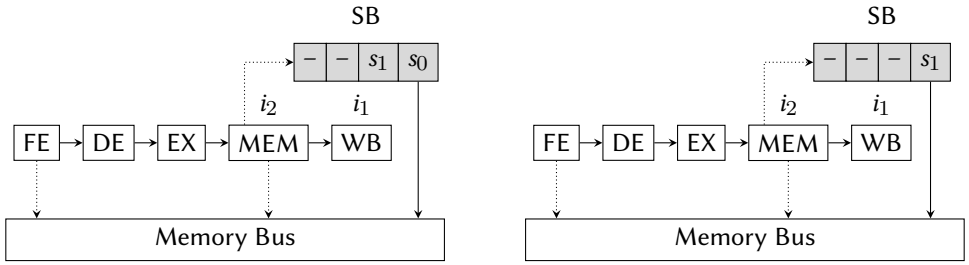
$$c.nlat(i) := \begin{cases} memlat_d(i) & : c.nstg(i) = MEM \wedge \neg dchit(i) \\ 0 & : otherwise \end{cases}$$

$$c.nstg(i) := \begin{cases} EX & : c.stg(i) = pre \\ MEM & : c.stg(i) = EX \\ WB & : c.stg(i) = MEM \\ post & : c.stg(i) = WB \end{cases}$$

$$c.ready(i) := c.cnt(i) = 0 \wedge (c.stg(i) = pre \Rightarrow \forall j < i,\ c.stg(j) \sqsupset_{\mathcal{S}} pre) \wedge (c.stg(i) = MEM \Rightarrow (opc(i) = store \Rightarrow \neg c.sbFull()))$$

$$c.free(s) := s = post \vee (\neg \exists j\ .\ c.stg(j) = s) \vee (\exists j\ .\ c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j)))$$

Figure 2.9: *cycle*() function for the simplified in-order 5-stage pipeline



(a) c'_a (b) c'_b

Figure 2.10: Monotonicity counterexample

The computation of the counter of remaining processing cycles $c.cnt(i)$ is performed as follows: for any instruction other than a load (and for a load in any stage other than MEM), the counter is decremented if it is positive, and a null counter remains equal to zero. For a load instruction inside the MEM stage, its counter is only decremented if the instruction performs a hit in the cache ($dchit(i)$) or uses the bus in the current cycle ($c.busTaken(i)$ is true).

Theorem 11

Store buffers jeopardize the monotonicity of in-order scalar pipelines.

Proof. We need to exhibit a counterexample where monotonicity is broken, that is to say a sequence of instructions i_1, ..., i_n and two pipeline states c_a and c_b such that $c_a \sqsubseteq c_b$ and $cycle(c_a) \not\sqsubseteq cycle(c_b)$. We start with the states c'_a and c'_b in Figure 2.10. In this figure, i_2 is a load instruction that requires the bus to access the memory. We assume that in c'_b the s_1 store has been initiated in the previous cycle and will use the bus at least for the next two cycles. As a result, $c'_b.busTaken(i_2)$ is false and $cycle(c'_b).cnt(i_2) = c'_b.cnt(i_2) = memlat_d(i_2)$. Conversely, we assume that s_0 will be completed at the end of the current cycle in c'_a. In the next cycle, s_0 will get out of the store buffer, and i_2 and s_1 will compete for the bus. Let us assume that i_2 gets the bus (i.e. $cycle(c'_a).busTaken(i_2)$). Now, if we rename $c_a = cycle(c'_a)$ and $c_b = cycle(c'_b)$, we have $c_a \sqsubseteq c_b$. From a less formal perspective that considers the stores inside the store buffer, c_a has made less progress than c_b, as s_1 has already started in c_b and not in c_a. Now, since s_1 is assumed to last for at least another cycle, $c_b.busTaken(i_2)$ is false, and the counter for i_2 will remain unchanged in $cycle(c_b)$. On the other hand, $c_a.busTaken(i_2)$ is true, so the counter for i_2 will decrease in $cycle(c_a)$. As a consequence, $cycle(c_a) \not\sqsubseteq cycle(c_b)$. □

2.3.2.2 Separate Store and Load Units

We provide the definition of the *cycle*() function in Figure 2.11. The main difference with the functions of the previous section concerns the topology of the pipeline that now includes separate SU and LU stages. In order to simplify the model, we consider only load and store instructions in this portion of the pipeline (we assume that other kinds of instructions are being directed to other parallel portions of the pipeline that include their corresponding functional units).

As before, the *pre* and *post* stages can model entire portions of the pipeline that are located respectively before and after the considered pipeline portion.

Theorem 12

Store buffers jeopardize the monotonicity of pipelines with separate store and load units.

Proof. The example of monotonicity violation is just a variation of the one we presented in Section 2.3.2.1, adapted to this particular topology. We consider states c'_a and c'_b as in Figure 2.12. i_1 is a load instruction that resides in stage LU. In state c'_a, there are two stores s_0 and s_1 residing in the store buffer. s_0 is currently being performed and will finish at the end of the cycle. We thus assume that the bus can be granted to the load unit in the next cycle. As a result, $cycle(c'_a).busTaken(i_1)$ is true. On the other hand, we assume a state c'_b in which the store buffer only holds s_1. We also assume that s_1 is currently being performed, and will use the bus for at least the next cycle. As a result, $c'_b.busTaken(i_1)$ is false, and will remain false for at least the next cycle. Consequently, $cycle(c'_b).busTaken(i_1) = false$. If once again we rename $c_a = cycle(c'_a)$ and $c_b = cycle(c'_b)$, we have $c_a \sqsubseteq c_b$ and since $c_a.busTaken(i_1)$, $cycle(c_a).cnt(i_1) = c_a.cnt(i_1) - 1$, while $cycle(c_b).cnt(i_1)$ remains unchanged. Finally, we have $c_a \sqsubseteq c_b$ and $cycle(c_a) \not\sqsubseteq cycle(c_b)$, so monotonicity is broken. □

2.3.3 Enabling timing-predictability with store buffers

In order to enforce timing predictability in the presence of store buffers, we propose to add a new gating mechanism that blocks load instructions in the stage before the one that actually performs the load (i.e. before MEM or LU in our examples) as long as the store buffer is not empty or there is a

$$\mathcal{S} := \{pre,\ S_p,\ SU,\ LU,\ post\}$$

$$pre \sqsubset_{\mathcal{S}} S_p \sqsubset_{\mathcal{S}} \{SU,\ LU\} \sqsubset_{\mathcal{S}} post$$

$$cycle(c)(i) := \begin{cases} (c.nstg(i),\ c.nlat(i)) & : c.ready(i) \wedge c.free(c.nstg(i)) \\ (c.stg(i),\ c.ncnt(i)) & : otherwise \end{cases}$$

$$c.ncnt(i) := \begin{cases} c.cnt(i) - 1 : c.cnt(i) > 0 \\ \qquad\qquad \wedge opc(i) = load \Rightarrow (c.stg(i) \neq LU \vee c.busTaken(i) \vee dchit(i)) \\ c.cnt(i) \quad : opc(i) = load \wedge c.stg(i) = LU \wedge \neg c.busTaken(i) \wedge \neg dchit(i) \\ 0 \qquad\quad : otherwise \end{cases}$$

$$c.nlat(i) := \begin{cases} memlat_d(i) & : c.nstg(i) = LU \wedge \neg dchit(i) \\ 0 & : otherwise \end{cases}$$

$$c.nstg(i) := \begin{cases} S_p & : c.stg(i) = pre \wedge opc(i) \in \{load,\ store\} \\ SU & : c.stg(i) = S_p \wedge opc(i) = store \\ LU & : c.stg(i) = S_p \wedge opc(i) = load \\ post & : c.stg(i) \in \{SU,\ LU\} \end{cases}$$

$$\begin{aligned} c.ready(i) := {} & c.cnt(i) = 0 \\ & \wedge (c.stg(i) = pre \Rightarrow \forall j < i,\ c.stg(j) \sqsupset_{\mathcal{S}} pre) \\ & \wedge (c.stg(i) = SU \Rightarrow \neg c.sbFull()) \end{aligned}$$

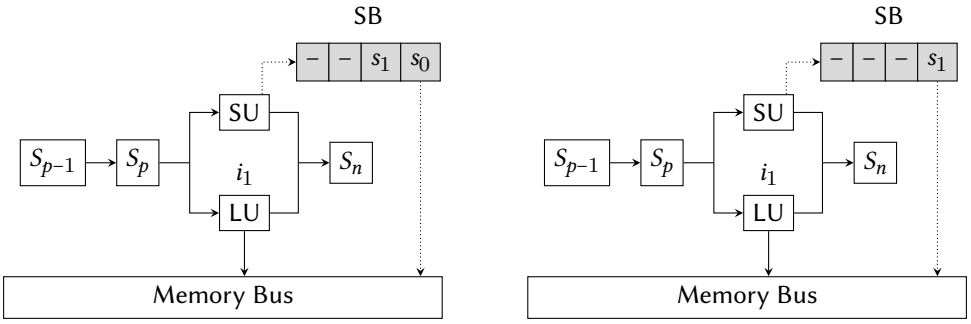
$$\begin{aligned} c.free(s) := {} & s = post \\ & \vee (\neg \exists j\ .\ c.stg(j) = s) \\ & \vee (\exists j\ .\ c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j))) \end{aligned}$$

Figure 2.11: *cycle*() function for the separate store and load units

store instruction in MEM or SU. The intuition behind this is to guarantee that the instructions acquire the bus following the program order, so that an access corresponding to an older instruction is always performed before an access corresponding to a younger instruction. In the models of Figures 2.9 and 2.11, this modification amounts to:

- adding a *c.sbEmpty*() predicate that is equal to 1 iff the store buffer and the SU (or MEM) stage are empty in state c;

- adding the following line to the *ready* function using a conjunction: $(c.stg(i) = S_p \Rightarrow (opc(i) = load \wedge c.sbEmpty()))$ (in which S_p is replaced by EX in the model of Figure 2.9).



(a) c'_a (b) c'_b

Figure 2.12: Monotonicity counterexample on a pipeline with separate store and load units

Theorem 13: Monotonicity is restored by the proposed gating mechanism in the in-order pipeline

Replacing the *ready* function of the pipeline model of Figure 2.9 by the following function restores the monotonicity in this pipeline:

$$\begin{aligned} c.ready(i) := c.cnt(i) = 0 \\ \wedge\, (c.stg(i) = pre \Rightarrow \forall j < i,\, c.stg(j) \sqsupset_{\mathcal{S}} pre) \\ \wedge\, (c.stg(i) = MEM \Rightarrow (opc(i) = store \Rightarrow \neg c.sbFull())) \\ \wedge\, (c.stg(i) = EX \Rightarrow (opc(i) = load \wedge c.sbEmpty())) \end{aligned}$$

Theorem 14: Monotonicity is restored by the proposed gating mechanism in the pipeline with separate store and load units

Replacing the *ready* function of the pipeline model of Figure 2.11 by the following function restores the monotonicity in this pipeline:

$$\begin{aligned} c.ready(i) := c.cnt(i) = 0 \\ \wedge\, (c.stg(i) = pre \Rightarrow \forall j < i,\, c.stg(j) \sqsupset_{\mathcal{S}} pre) \\ \wedge\, (c.stg(i) = SU \Rightarrow \neg c.sbFull()) \\ \wedge\, (c.stg(i) = S_p \Rightarrow (opc(i) = load \wedge c.sbEmpty())) \end{aligned}$$

Both theorems have been proven using the Coq proof assistant. The Coq proofs are available online, and the main elements of the proofs are presented in a paper that we published in RTNS'23 [35].

2.3.4 Implementation in MINOTAuR

We implemented our proposed mechanism in the MINOTAuR RISC-V core [33] to evaluate its impact on performance. As we saw earlier, in MINOTAuR, the load/store unit is pipelined: in its second stage, stores are handled by the store unit, and loads by the load unit. The memory hierarchy of MINOTAuR is depicted in Figure 2.13, that can be seen as a more detailed zoom on the memory section of the complete MINOTAuR pipeline depicted in Figure 2.1.

This architecture is a superset of the model described in Figure 2.11. The data cache acts as an intermediary between the core and the memory bus. The mechanism used to send and remove

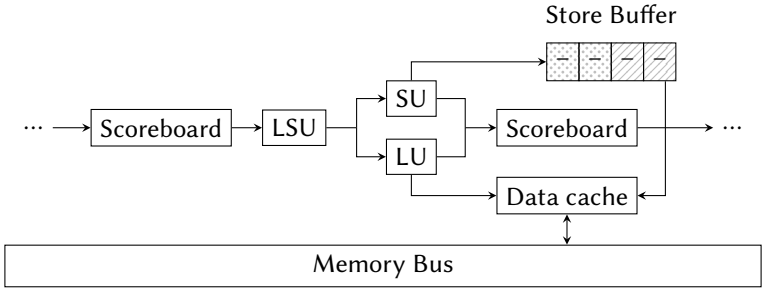


Figure 2.13: Memory components in the MINOTAuR core

commands from the store buffer, which is abstracted in the model, is described in the remainder of this section.

2.3.4.1 The store buffer

The store buffer is split into two queues. Store commands are inserted into the first one (the "speculative queue") by the SU, and remain there until the corresponding instruction is committed. When an instruction is committed, the corresponding command is moved to the second queue (the "commit queue"), and remains there until it can be sent to the memory hierarchy.

In the RV32 ISA, a write can be performed on 8, 16 or 32 bits. Hence, the buffer retains the exact physical address and the size of the write operation. When a request is issued to the data cache, the data and the address are realigned to a 64-bit block. Once the request has been acknowledged, the command is removed from the buffer.

Before issuing a fetch request, the LU checks if there is a write pending to the same 64-bit block in the store buffer. In this case, the load will be put on hold in the LSU until the write has been submitted and acknowledged by the cache.

2.3.4.2 The data cache

In MINOTAuR, the data cache is write-through (data are written in the cache and the main memory at the same time), and does not allocate entries on writes (if a line to be written is not cached, it is not loaded). The data cache does not immediately forward requests from the store buffer to the bus, but stores them in a write buffer. Its purpose is to reduce the amount of requests performed on the bus by merging pending writes to the same memory block. As part of our gating mechanism, we make sure that the LU stalls if there is a store pending in this buffer, as well as in the store buffer. For the sake of simplicity, we do not describe its exact operation, but it does not affect the validity of our proofs, nor the monotonicity of the pipeline.

The cache memory receives requests from the load unit, and can answer in a single cycle if the line is already cached. It can also read from the write buffer: if the LU requests a line awaiting to be written, the cache will combine the dirty bytes in the buffer and the line in the memory.

2.3.4.3 Modeling the gating mechanism for MINOTAuR

The formal model that we provided in Figure 2.3 already includes the gating mechanism that we described in Theorem 14. Looking at the formula of Theorem 14, we are only interested in the terms that concern SU and Sp. The $(c.stg(i) = SU \Rightarrow \neg c.sbFull())$ term is covered by the $(s \in$

$\{\text{IF}, \text{IS}, \text{LSU}, \text{SU}\} \wedge c.slot(s))$ term in the definition of the $c.free(s)$ function in the model of Figure 2.3, since the definition of $slot(\text{SU})$ models the store buffer occupation. Now, the $(c.stg(i) = S_p \Rightarrow (opc(i) = load \wedge c.sbEmpty()))$ term is covered the $c.stg(i) = \text{LSU} \Rightarrow (...) \vee (opc(i) = load \wedge (\neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))$ term in the definition of the $c.ready(i)$ function of Figure 2.3. Indeed, $\neg c.pending(i, store)$ states that no store operation resides between the LSU and the ST stages, which means that the store buffer is empty.

We thus safely conclude that the complete model of MINOTAuR that we proposed in the previous section includes the proposed gating mechanisms to ensure the monotonicity of progress in the presence of a store buffer. This is in fact a reason why we managed to prove the monotonicity in Section 2.2.2.

In the next section, we present the experiments we conducted in order to evaluate the performance cost of our modifications of the Ariane core in order to turn it into a provably timing predictable core.

2.4 Evaluation

2.4.1 Organization of the evaluation section

In this section we report all the experimental results we obtained regarding MINOTAuR and the various mechanisms that we implemented in order to achieve performance while ensuring timing predictability. The first experiments (Section 2.4.2) were conducted on Ariane$^+$ and on our first version of MINOTAuR that was directly derived from it. These results show that unlike what was reported by Hahn et al. in [40], entirely removing the speculative execution mechanisms in Ariane$^+$ has a strong negative impact on performance, and thus validates our efforts to combine predictability and speculative execution to maintain a higher level of performance. We also measured the implementation cost and performance impact of the back-up mechanisms of Section 2.2.3 on MINOTAuR. Finally, we measured the performance impact of the gating mechanisms that we introduced in Section 2.3.3 regarding the store buffer (Section 2.4.3).

2.4.2 Evaluation of MINOTAuR and of the back-up mechanisms

2.4.2.1 Methodology

All our extensions have been implemented in the SystemVerilog model of the Ariane$^+$ core and our processors have been synthesized with Xilinx Vivado 2021.1[3], targeting a Xilinx Zynq XC7Z020-1CLG400 on a Digilent Zybo Z7-20 board, with the `PerformanceOptimized` directive set[4]. The memory has a latency of 11 cycles.

We have used the `kernel` and `sequential` sets of programs of the TACLe benchmark suite[5] [38] as well as CoreMark[6] as benchmarks, all compiled with `gcc 10.2.0`[7] and optimization flag `-O2` (`-O3` for CoreMark).

The results are displayed in Table 2.1. We report the arithmetic mean of the overheads. Since the number of cycles taken to execute the benchmarks varies from a few hundreds to a few hundred millions, we also computed a global overhead (corresponding to the overhead between the total cycles of the various processor variants) and the difference between the geometric means of the number of cycles in the variants over the TACLe benchmarks. The overheads displayed in the first three rows of

[3] `https://www.xilinx.com/products/design-tools/vivado.html`

[4] Except for 16 backups/RAS-16, which would not fit on our FPGA with this configuration

[5] We had to exclude `mpeg2` which did not compile, and `susan` which failed to execute, both due to memory exhaustion on Ariane$^+$, and MINOTAuR.

[6] `www.coremark.org`

[7] `https://github.com/riscv-collab/riscv-gnu-toolchain/tree/ed53ae7`

the table (marked with *) are computed w.r.t. Ariane$^+$, while the rest of the rows (marked with †), which correspond to variants of the back-up mechanism implemented on top of MINOTAuR, display overheads w.r.t. MINOTAuR.

The source code for all cores and experiments presented in this chapter is available in [32].

2.4.2.2 Results

We started by applying the gating mechanism of Hahn et al. [40] to the Ariane$^+$ core. This mechanism is more stringent than the one we described in Figure 2.3, as it completely precludes speculative execution. Our objective was to measure the cost of this method on a more complex processor than the SIC.

As expected, we did no longer observe any inversion. Compared to the baseline Ariane core (cf. row Ariane$^+$ + SIC in Table 2.1), the overhead in execution cycles amounts to 38.96% on average (with a geometric mean of 37.73%), and the global overhead reaches 45.68%. These results are significantly higher than the 6-7% loss reported in [40]. We believe that this may be related to the fact that Ariane$^+$ is much more advanced than the 5-stage in-order pipeline upon which the SIC processor was designed. In particular, Ariane$^+$ includes dynamic branch predictors (Hennessy and Patterson [46] report a 30% performance gain using such predictors) and several queues that allow some instruction parallelism. For example, the scoreboard (modelled by the *iqueue*) makes it possible, to some extent, to execute several instructions in parallel in different functional units.

The results for MINOTAuR are also displayed in Table 2.1. Again, we did not observe any inversion, which was expected due to the gating mechanism that we have implemented. Since we carefully selected the restrictions that were absolutely required to prove timing predictability and relaxed the other ones, the performance loss compared to the Ariane$^+$ core is noticeably low: 1.81% on average, with a global overhead of 0.69% and a difference between the geometric means of 1.65% only. By relaxing the limitations on speculative execution, we thus claimed back more than 35% of the performance on average (compared to Ariane$^+$ + SIC), while keeping the core provably timing predictable. Small benchmarks tend to have higher overheads than large ones. We believe this is due to the warming of the caches and the initial filling of the pipeline: the temporal impact of cache misses and of an empty pipeline is proportionally higher when the application consists of only a few hundred instructions. The cost of timing predictability in terms of performance in MINOTAuR is thus negligible. We even remark that some benchmarks run faster on MINOTAuR than on Ariane$^+$: by preventing speculative fetches of instructions from the main memory, we prevent the pollution of the instruction cache during speculative executions that will eventually be discarded, thus reducing the number of instruction cache blocks that need to be re-fetched after a wrongly speculated branch.

Let us now focus on the results of the second part of Table 2.1. We first evaluated the efficiency of our backup mechanisms. To do so, we equipped MINOTAuR with a LRU instruction cache instead of its orginal random one, and set the size of the RAS to 16 (thus allowing fast returns for up to 16 nested function calls), and varied the size of the RAS and LRU backups. All the performance results of the core variants are compared to the baseline MINOTAuR that is still equipped with a random instruction cache and no RAS.

We first see that using 2-slot buffers for the backup mechanism yields slightly worse results (0.39% overhead on average) than the baseline MINOTAuR that has no RAS and random caches. This is due to the fact that the LRU instruction cache backup mechanism blocks all instructions (including the ones resulting in a hit) as soon as there are at least 2 pending branch instructions in the pipeline, while the random cache of the baseline MINOTAuR does not. The versions in which the number of backup slots is 4 or more all perform better than the baseline MINOTAuR. The versions with 8-slot and 16-slot backup perform equivalently.

Table 2.1: Resource usage, CoreMark score and average overhead for Ariane$^+$ and multiple MINOTAuR variants.

Core	LUTs	Max freq.	CoreMark	Total cycles	Arith. mean	Geo. mean	Global overhead	Deviation
Ariane$^+$	17,106	34.93 MHz	110.36	599,217,366				
Ariane$^+$ + SIC *	14,754	32.11 MHz	72.03	872,935,255	38.96%	37.73%	45.68%	0.196
MINOTAuR *	17,176	32.97 MHz	110.58	603,373,808	1.81%	1.65%	0.69%	0.059
2 backups/RAS-16†	21,782	30.40 MHz	108.78	607,374,373	0.39%	0.37%	0.66%	0.021
4 backups/RAS-16†	23,952	30.46 MHz	110.90	591,280,403	-1.28%	-1.29%	-2.00%	0.014
8 backups/RAS-16†	26,550	29.36 MHz	110.90	590,971,752	-1.32%	-1.33%	-2.06%	0.014
16 backups/RAS-2†	19,439	33.83 MHz	110.89	591,998,705	-1.22%	-1.33%	-1.89%	0.014
16 backups/RAS-4†	22,012	31.62 MHz	110.90	590,988,103	-1.31%	-1.23%	-2.05%	0.014
16 backups/RAS-8†	26,175	31.46 MHz	110.90	590,971,431	-1.36%	-1.32%	-2.06%	0.015
16 backups/RAS-16†	–	–	110.90	590,971,205	-1.32%	-1.37%	-2.06%	0.014

* the Arith. mean, Geo. mean, Global overhead and Deviation are computed w.r.t. Ariane$^+$
† the Arith. mean, Geo. mean, Global overhead and Deviation are computed w.r.t. MINOTAuR

Then, we evaluated the impact of the RAS size on the performance of the core. Enabling the RAS with a size of more than 2 yields better results than having no RAS, at the expense of resources on the FPGA. The only model we tested that had worse performance than the baseline MINOTAuR was the 2-slots buffers for backups and 16 entries in the RAS. Since all other variants we tested perform better, including those with a smaller stack, we can conclude that the RAS is not the cause of this result.

Overall, the average gains induced by the RAS and LRU caches are around 2% which is not a significant improvement. However the use of an LRU cache instead of a random one has a huge impact on the precision of the static analyses and in turn on the precision of the WCET. In the light of these results, it seems that a reasonable trade-off between performance, predictability, LUT consumption and maximum achievable frequency is the version with 16 levels of backup and a RAS of size 2, which yields a 13% increase in the LUT usage, but also a lower number of execution cycles than the baseline MINOTAuR. We display the maximum achieved frequency for each design in the table, as an indication. However, the frequencies do not seem correlated to the complexity of the designs. This indicates that the observed reductions in maximum frequency are not due to a lengthening of the critical path, but rather to the small size of our FPGA that prevents mapping and routing optimizations by the synthesis compiler. Finally, based on these benchmarks, it seems that the cost of the RAS (in terms of LUT usage) is difficult to justify. This is due to the fact that TACLe benchmarks do not contain enough nested function calls to gain much advantage from it.

2.4.3 Evaluating the cost of the store buffer gating mechanism

We now compare the performance of the MINOTAuR core with and without the gating mechanism described in Section 2.3.3, to assess the cost of timing predictability regarding the store buffer. Additionally, we also implemented a more aggressive version of our gating mechanism in which load instructions are stalled directly in the LU stage. We believe that this mechanism also enforces the monotonicity of the pipeline, but have not proved it yet[8].

[8]We focused instead on the general model and proofs that we presented, whereas the gating at the LU stage relies on implementation details that belong to MINOTAuR, which would have made the proofs too specific.

Table 2.2: LUT usage, CoreMark score and performance on the TACLe benchmarks

	Gating mechanism		
	Disabled	Enabled	Gating at LU
LUTs	17,145	17,153	17,155
CoreMark score	110.44	108.67	110.20
Total cycles	603,439,496	606,237,172	605,359,342
Total cycles at `-O0`	1,202,647,474	1,225,049,748	1,216,564,217

2.4.3.1 Experimental setup

The modifications were made to the SystemVerilog description of MINOTAuR, synthesized with Xilinx Vivado 2021.1 for a Xilinx Zynq XC7Z020-1CLG400 on a Digilent Zybo Z7-20 board, with a frequency of 25 MHz, and a memory latency of 11 cycles. All the results presented in this section correspond to actual measurements performed on the FPGA, running either CoreMark or the TACLe[9] [38] benchmark suite compiled with `gcc 10.2.0`, respectively at optimization level `-O3` and `-O2`. We also ran the TACLe benchmarks at optimization level `-O0`, to prevent `gcc` from improving the memory usage patterns, and thus potentially hiding part of the cost of our changes.

We report various measurements for our cores: their LUT (Look-Up Table) usage, their CoreMark score, the number of cycles taken by the 50 programs from the TACLe benchmark suite, the arithmetic and geometric means of the overheads in MINOTAuR induced by our gating mechanism, as well as the total cycles overhead.

Table 2.3: Overheads for the gating mechanism on TACLe

	Gating at LSU		Gating at LU	
	At `-O2`	At `-O0`	At `-O2`	At `-O0`
Arithmetic mean	2.07%	2.89%	1.53%	0.88%
Geometric mean	2.00%	2,85%	1.49%	0.85%
Global overhead	0.46%	1.86%	0.32%	1.16%
Minimum	-4.61%	-2.78%	0.00%	0.00%
Maximum	17.74%	10.07%	12.40%	14.47%

2.4.3.2 Results

Table 2.2 shows that our gating mechanism does not significantly increase the resource usage of the core: 8 LUTs are added to the 17,145 LUTs of the original design.

On average, our gating mechanism results in a loss of performance of 2.07% on the TACLe benchmarks, as reported in Table 2.3. On individual benchmarks, the performance loss is inconsistent: some benchmarks, such as `bitonic`, are not significantly affected by our gating mechanism, but programs with different memory access patterns (i.e. `md5`, `sha`) are more impacted by this change. A few benchmarks, such as `h264_dec`, are instead significantly faster. We also see that performing the gating in the LU stage yields slightly better average results than performing it in the LSU stage (around 0.5% improvement).

[9] Once again, `mpeg2` and `susan` were excluded because they failed to compile (resp. execute) due to memory exhaustion.

At optimization level `-O0`, our mechanism has a higher cost (3% instead of 2% in average, and 2% instead of 0.5% overall), but its impact remains low, even though memory access patterns are not optimized.

2.5 Conclusion

This chapter presented issues caused by instruction level parallelism regarding the timing predictability of processor pipelines. These issues appear when two or more instructions concurrently request a shared sequential resource and may lead to timing anomalies, thus preventing safe and scalable static WCET analyses. We presented our work on the MINOTAuR core, a timing predictable version of the Ariane RISC-V core. This work was mainly inspired by the work of Hahn et al. on the SIC processor [40]. However, we showed that on a more complex core such as Ariane, precluding the speculative execution altogether has a huge impact on performance, and thus that speculation should be tolerated when it does not break the predictability of the core. We proposed a series of generic modifications to obtain a core that is provably timing predictable while maintaining a level of performance close to the baseline core, and used Ariane as a basis for demonstration. The first modification regards concurrent accesses to the memory bus between the instruction and data caches. We made that modification different from what existed in the state-of-the-art, in order to tolerate speculative execution, and proved that under some restrictive hypotheses, the modified core was timing predictable. We then proposed some back-up mechanisms to make our solution compatible to any core using a cache with an aging mechanism and a RAS, thus lifting the restrictive hypotheses. Finally, we took interest in store buffers and the contentions they can generate with load instructions in the data cache. Once again we proposed a generic gating mechanism to preclude concurrent accesses that could impair the timing predictability of the pipeline, and then used MINOTAuR to measure its impact on performance.

Regarding the contents of this chapter:

- the work was originally part of the master 2 internship of Alban Gruin (2021), and then of his ongoing PhD thesis work (2021-). The thesis is directed by Pascal Sainrat and myself and advised by Christine Rochange as well, and is funded by a grant from the ministère de l'enseignement supérieur et de la recherche. The master 2 internship was funded by Labex Cimi.
- the results were published in RTSS'21 [33], IEEE Transactions on Computers [34] and RTNS'23 [35]. The RTSS'21 paper was awarded an outstanding paper award.
- a validation method for the model of cores, based on automatically-generated simulators, has been developed and presented in a paper published in the WCET'23 workshop [36].
- all the proofs have been written in Coq by Alban Gruin. The proofs of Section 2.3.3 have been published in the RTNS'23 paper. The Coq version of the proofs of Section 2.2.2 were performed recently, and a paper is currently being written to publish them.
- our results with MINOTAuR have led to the submission and acceptance of the ANR PRC ProTiPP project (2023-2026) led by Christine Rochange, that regards processor models and proof automation, and of the ANR ASTRID PRINTEMPS project (2024-2027) led by Pascal Sainrat, with industrial partners at Thales Research and Technology, regarding the implementation of a predictable and secure multi-core processor based on the MINOTAuR core.

Chapter 3

Multi-core timing analysis with the multi-phase model

3.1 Introduction

The growing adoption of multi-core processors in industrial real-time systems [67, 68] raises the challenge of providing safe and tight Worst-Case Execution Time (WCET) bounds for tasks running in parallel on separate cores. Indeed, in multi-core architectures, the cores execute their processes/threads independently from one another, but they share some hardware components such as caches, buses and memories. Contentions may happen in these shared components: when a task requires to access a component which is already in use by another task running on another core, it has to wait until the component is free again. This phenomenon incurs execution delays which depend on the context of the task execution (which other tasks are running in parallel, and are they accessing the shared resources ?). In traditional single-core WCET analysis [1, 9], each task is analysed in isolation i.e. as if no other task was running in parallel. Then a schedulability or Worst-Case Response Time (WCRT) analysis is performed using a model in which each task is represented by its WCET, in order to guarantee that each individual task meets its deadline or that the system as a whole meets an end-to-end timing constraint. A direct consequence of the delays incurred by inter-core contentions is that traditional WCETs no longer represent a safe upper-bound on the execution time of the tasks when they are run on multi-core processors. It becomes necessary to model tasks using at least their WCET in isolation and their worst-case number of accesses to shared components, and to perform an additional analysis to safely upper bound the interference effect of the potential contentions. However, this classical model, which maps one task to one temporal phase was not designed with multi-core interference analysis in mind, and may not be the best-suited to analyse tasks running in parallel.

More recent models represent each task as a sequence of *phases*, each characterized by a WCET and a number of accesses, either as an attempt to increase the precision of the interference analysis [64, 3], or in order to build schedules in which there is no interference [27, 66]. This *multi-phase* abstraction maps temporal phases to actual sections of code that are separated by synchronizations.

In the remainder of the chapter, we focus on memory accesses as the sole source for interference in the system. We assume a multi-core target with shared memory and a sequential First-Come-First-Served memory bus, in which each core is equipped with a private L1 data cache and a private scratchpad to hold the instructions of the tasks it executes. However, the abstractions that we describe naturally support any other kind of interference source, and generalize to other architectures (e.g. L1 instruction cache): the only thing that changes is the analyses that must be performed on the code in order to obtain the abstract models of the tasks. In particular, we consider non-preemptive static

(or fixed-priority time-triggered) scheduling, but limited preemptions could be supported by adapting classic cache-related preemption delay (CRPD) computation techniques.

3.1.1 Related Works

The real-time systems community has been working on the problem of multi-core interference for nearly two decades now. A comprehensive survey on the topic has been published in [54]. In this section we position our work within the state-of-the-art, and focus on two existing analysis frameworks for which our results can be particularly useful.

Reduction of Interference Through Predictable Execution: The model we present here can be seen as a generalization of the PRedictable Execution Model (PREM) [63] for multi-core architectures, or as a relaxation of the constraints of the Acquisition-Execution-Restitution (AER) [27, 66] execution model. The original idea of PREM was to avoid interference between memory accesses and asynchronous I/O traffic on a bus by carefully scheduling and enforcing the execution of tasks so that it does not occur in parallel with I/O interrupts or DMA transfers. The Time Interest Points (TIPs) framework that we present in Section 3.4 leverages this idea to the problem of multi-core interference analysis: the primary objective is to generate timing and memory access profiles of real-time tasks in order to statically schedule them on multi-core processors in a way that carefully accounts for, and possibly reduces the interference between them. The AER execution model aims at suppressing all interference by construction. The idea is to separate the execution of each task into three consecutive parts: the acquisition (A) of code and data for the task, the execution (E) of the task, and the restitution (R) of the outputs of the task to the shared memory. This separation is ensured either by the programmer or by the compiler [61]. Then the tasks are statically scheduled in a way that ensures that the A and R parts of the different tasks never occur in parallel. The TIPs framework implements the same idea, but the granularity at which it works (single memory accesses) is much finer, and it does not require to compile the task as three separate parts. This has multiple advantages such as the possibility to analyse and deploy legacy code with only small, automatic modifications (for synchronizations), and the limitation of the memory overhead due to static reservation in the AER model. Another difference is that TIPs allow the construction of programs in which some amount of interference can be tolerated (and statically quantified for compositionable processors [39]).

WCRT Analysis Frameworks: In [20] the authors present a WCRT analysis framework for sporadic task systems scheduled on multi-core processors using a preemptive fixed priority algorithm (and static partitioning of tasks on the cores). The authors consider that each possible execution trace of each task in the system is available for analysis, and from this set provide precise formulas to quantify the effect of interference between tasks on the shared elements of the target processor (memories, busses, processor time). This work extends classical WCRT analyses [49] by introducing new interference terms to cover the particularities of multi-core processors, and by making it possible to precisely account for the execution context of the tasks (i.e. which other tasks are running on the same core, or in parallel). These terms are computed by extracting worst-case information for any time interval of any given size on the execution traces of tasks. In [20] the authors discuss the empirical complexity of obtaining and manipulating the entirety of the execution traces for a task system corresponding to an industrial application. Their conclusion is that traces are a desirable abstraction of the tasks execution behavior since they can be easily manipulated and they express precisely the relation between the task and the shared resources. In particular they emphasize the fact that the worst case behavior of a task depends on its execution context, and that traces allow to exploit this.

They conclude that although working on all execution traces is unfeasible for arbitrary applications, it is possible to feed the framework with a set of abstract traces which overestimate the worst case behaviors of the tasks. However nothing is said on how to obtain such an abstraction, nor on the potential costs of the various abstraction methods that could be used.

In [64] the authors provide a method close to real-time calculus [73] in order to compute the WCRT of a task system on a multi-core processor. Each task is represented as a sequence of time intervals, and for each time interval, a bound on the worst case number of memory accesses performed by the task is assumed to be known. Using this information, memory access arrival curves are derived and then combined to upper-bound the interference effect in time. A method is briefly sketched to derive the time intervals, which assumes precise knowledge on the tasks behavior (in particular local best and worst case execution times), but nothing is said on how this knowledge can be acquired in practice, nor on the abstraction cost of building the time intervals this way.

3.1.2 Chapter organization

The remainder of the chapter is organized as follows: in Section 3.2 we start with simple examples to introduce in an intuitive fashion the main concepts that will be used throughout the chapter. Then in Section 3.3, we provide a formal framework to describe the multi-phase model as well as criteria that guarantee the correctness of a multi-phase implementation with regards to the interference analysis. Section 3.4 then describes the Time Interest Points framework, which we designed in order to obtain multi-phase representations of tasks from their binary code. We then present static scheduling techniques for the multi-phase model in Section 3.5, and conclude the chapter.

3.2 Introductory examples

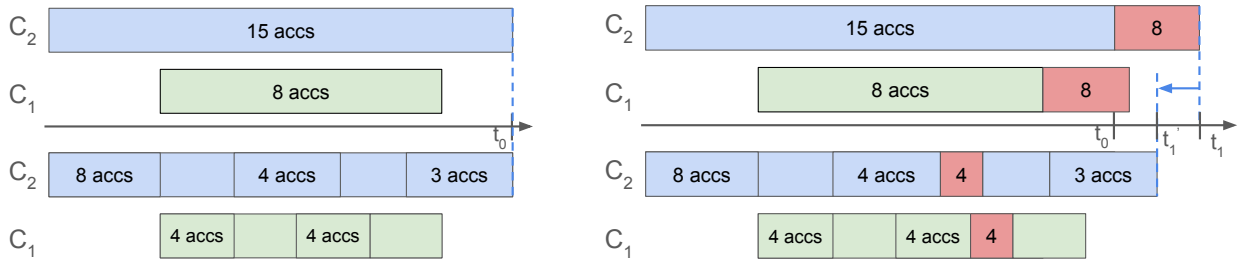
In this section we provide a quick and intuitive introduction to the multi-phase model, first by showing how it can be beneficial to the precision of the interference analysis, and then by illustrating the main technical concepts of the chapter (phases, abstract execution traces and synchronizations) using a simple example.

3.2.1 Interference analysis and the multi-phase model

We start with the example of Figure 3.1. In this figure we represent a blue (resp. a green) task scheduled on core C_2 (resp. C_1). In Figure 3.1a, we display the static reservation of the cores for the two tasks, before performing an interference analysis. On top, both tasks are represented as a single phase, characterized by a WCET and a worst-case number of accesses. At the bottom, the same tasks are represented in the multi-phase model: the execution of the blue task is seen as a sequence of 5 phases that perform respectively 8, 0, 4, 0 and 3 accesses. The green task is composed of 4 consecutive phases that perform 4, 0, 4 and 0 accesses.

During the interference analysis, the objective is to upper-bound the number of contentions that each task can be subjected to, and to multiply this number by a timing penalty bound. The results are displayed in Figure 3.1b. On top, the analysis is performed at the granularity of each task: out of the 15 accesses of the blue task, at most 8 can be interfered by accesses from the green task, resulting in an additional time reservation shown in red. In the same fashion, in the worst case, the 8 accesses of the green task can be interfered by 8 of the 15 accesses of the blue task. This results in the blue task finishing its worst-case execution at date t_1. On the other hand, at the bottom, the interference analysis is performed at the granularity of the phases: since the first phase of the blue task is scheduled to execute with no other phase in parallel, its 8 accesses are guaranteed to happen without interference.

The same is true for the 4 accesses of the first green phase, as it is scheduled in parallel with a blue phase that performs no access. The third green and blue phases both perform 4 accesses and are scheduled partially in parallel, so we account for 4 contentions for each. The remainder of the phases are guaranteed to run free from interference. In the end, by using a finer-grain representation of the profile of accesses in time, the worst-case number of contentions is reduced to 4 for each task. This results in a reduced worst-case end time for the green and blue tasks.



(a) Before interference analysis

(b) After interference analysis

Figure 3.1: Single- vs Multi-phase representation of two tasks before and after interference analysis

3.2.2 Traces, phases and synchronizations

Before we describe the formal model of the multi-phase representation and of the abstract execution traces that are used to correctly account for memory accesses in time, we briefly introduce these elements and their mutual relationship using the example of Figures 3.2 and 3.3. In the right part of Figure 3.2, we display the disassembled ARM assembly code of a short program. The colored instructions correspond to memory loads and stores that have been characterized as potentially resulting in a cache miss, and thus that may generate or be delayed by contentions in the memory subsystem. Note that in this particular example, the colored instructions were arbitrarily selected for illustration purposes. The left part of the figure displays 3 abstract execution traces. Each node (colored circle) represents the corresponding colored memory instruction in the code, and each arrow represents the worst-case execution duration between its source and destination nodes. The first three accesses (blue, purple and green) are performed in all traces at dates 0, 157 and 257 in the worst case. Indeed the control flow of the program always includes the corresponding instructions. Then, the first trace accounts for the red access at date 463 in the worst case, while the second trace accounts for the execution of the yellow access at date 618 and the last trace reaches the end of the program with no additional access. Each path in the control flow of the program (and thus each actual execution trace) is thus covered by an abstract trace. For each trace, we depict the actual timing of one possible execution. Each memory access is marked by a cross at the date at which it happens, and a dashed line links the cross to its corresponding node.

Without additional information from the model, one must consider that each memory access can be performed at any time before its worst-case date. As an example, we depict the time windows for which the red (resp. yellow) access must be accounted for as a red (resp. yellow) striped rectangle. If we look at the multi-phase representation at the bottom of the figure, it means that for each phase, all accesses whose worst-case date is after the start of the phase must be accounted for in this phase. For the first phase on the left, the four accesses of the top trace have an execution date superior or equal to the start date of the phase, so all four accesses must be accounted for in this phase. The same is true for the middle trace. The bottom trace only performs three accesses, which must also

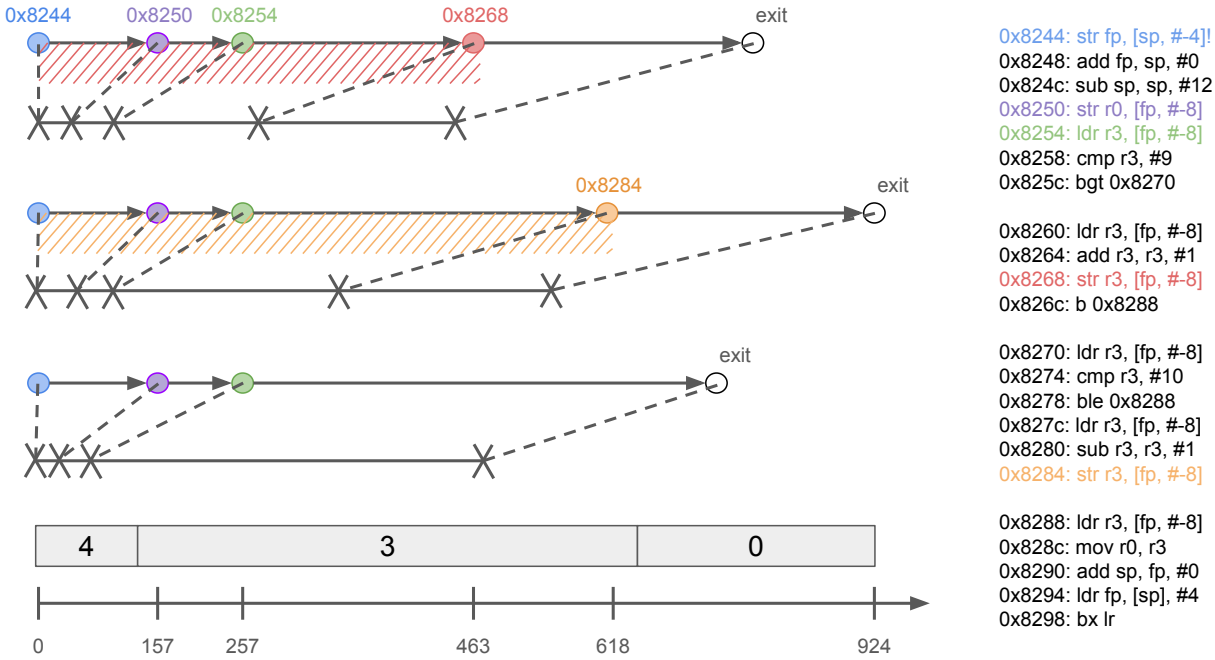


Figure 3.2: Abstract execution traces, corresponding concrete executions, and multi-phase representation

be accounted for in the first phase. Now, since at most one trace is executed at a time, we account for the maximum number of accesses by a single trace: 4. The worst-case execution date of the blue access is 0, so there is no possibility that it happens in the second phase. However, according to the model all other accesses can happen in the second phase, so we account for 3 accesses in the phase.

This incapacity to lower bound the execution date of a memory access leads to an over-estimation of the number of accesses in the phases, as some accesses are accounted for in multiple phases. In our example, the sum of the accesses in the three phases amounts to 7, while if we used the single-phase model, we would only have accounted 4 accesses in the worst case for the whole task execution. In order to reduce this imprecision, one could compute the best-case execution time (BCET) of the memory accesses. We choose to use another solution, mainly for two reasons:

- BCET analysis adds imprecision to the model. In particular, it is likely that the BCET of a particular access is less than the WCET of one or more of its predecessors, leading us to still account for two or more accesses during the intersection of their respective [BCET, WCET] time windows.
- Once the system is scheduled and the interference analysis is performed, it is crucial to enforce that each access can only occur in the time window in which it was accounted for during the analysis (see Section 3.3.4 for more details). BCET analysis does not offer this level of control with a satisfactory timing granularity.

We instead rely on time synchronizations. Some nodes in the abstract execution traces are selected to be synchronized: the corresponding instructions are not allowed to execute before a particular date that is determined statically. When a node is synchronized inside a phase, its accesses are thus guaranteed not to occur in the previous phases. This property transitively extends to the successors of the

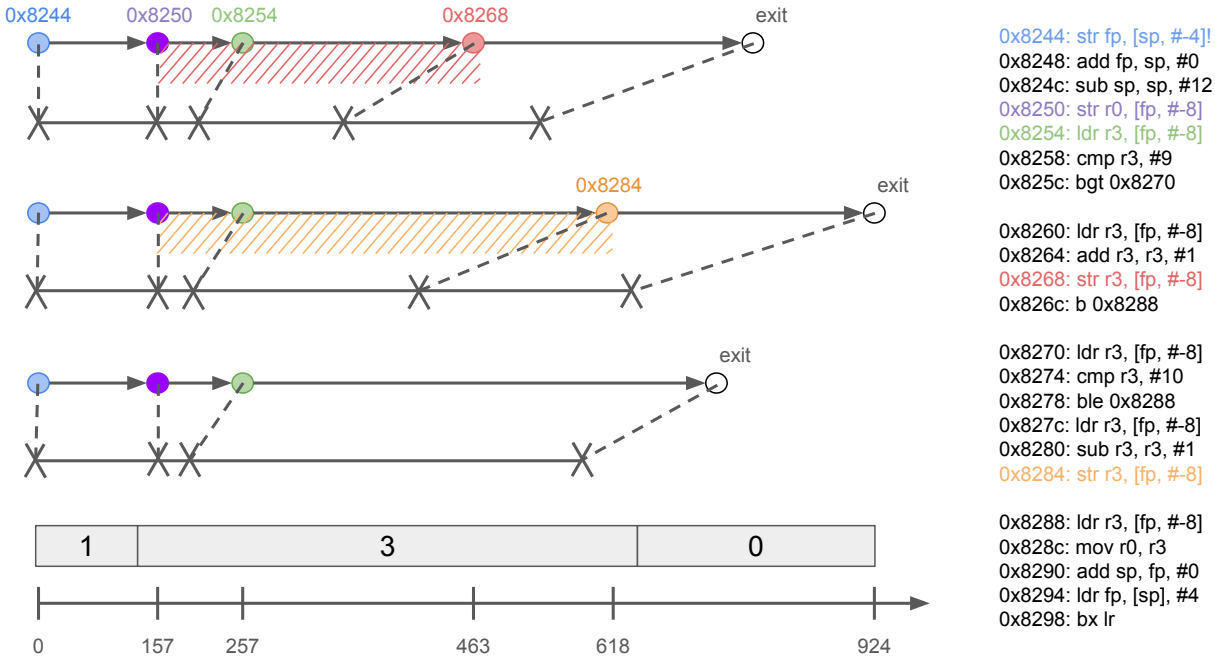


Figure 3.3: Abstract execution traces, corresponding concrete executions, and multi-phase representation, when a synchronization is added

node. An example is shown in Figure 3.3. The purple instruction at 0x8250 has been selected for synchronization at its worst-case date (depicted with dark purple). As a consequence, the corresponding access can only be initiated at date 157 in any trace. It follows that this and all subsequent accesses cannot occur in the first phase: we only need to account for the blue access in this phase, and the overestimation is reduced task-wise. Compared to Figure 3.2, the actual access corresponding to the purple node is now postponed at date 157 on each trace, and the spans of the red and yellow nodes now start at date 157 instead of 0.

3.3 The multi-phase model

3.3.1 Task models

We model a system of real-time tasks τ^i ($i \geq 0$). Each task is represented in two separate ways, as depicted in Figure 3.4:

- a "time-centric" representation called *multi-phase*. In this abstraction, the task is modelled by a sequence of time slots, called *phases*, which covers its WCET. We call this sequence of phases a *profile*. Each phase is associated to an upper bound on the number of memory accesses that the task can perform during the corresponding time slot. This representation is used to statically compute the schedule and perform the interference analysis of the system (in a timing-compositional approach). The mapping between tasks and multi-phase profiles is not bijective: multiple profiles can be found that represent the same task.
- a "code-centric" representation. In this abstraction, a task is represented by all its possible

Notation	Definition
τ^i	task i
ϕ^i_k	phase k in the representation of τ^i
$\phi^i_k.d$	start date of ϕ^i_k without interference
$\phi^i_k.dur$	worst-case duration of ϕ^i_k without interference
$\phi^i_k.m$	maximum number of memory accesses performed within ϕ^i_k
t^i_j	execution trace j of task τ^i
$\eta^i_{j,k}$	node k in trace t^i_j
$\eta^i_{j,k}.it$	instruction represented by $\eta^i_{j,k}$
$\eta^i_{j,k}.d$	worst-case execution date of $\eta^i_{j,k}$ without interference
$\eta^i_{j,k}.m$	maximum number of memory accesses performed by $\eta^i_{j,k}$
$\eta^i_{j,k}.sync$	True if the node is synchronized, i.e. cannot be executed before its $\eta^i_{j,k}.d$
$s_{last}(\eta^i_{j,k})$	last synchronized node before $\eta^i_{j,k}$ in trace t^i_j
$t^i_j\|_{\phi^i_k}$	restriction of trace t^i_j to ϕ^i_k, i.e. the set of nodes in t^i_j that may execute during ϕ^i_k

execution traces (i.e. all the possible sequences of instructions executed from the start of the task to its end). Since this set may be too large to analyze in practice, we consider memory-centric traces: only instructions which may perform memory accesses[1] are represented in the traces, and the rest of the instructions is abstracted by computing local WCETs. This representation is an intermediate step to go from the binary code of a task to its multi-phase representation, and back: it allows the number of memory accesses in each phase to be bounded correctly, and to insert synchronization code at the correct locations in the binary to enforce the scheduling choices.

We denote $\mathbb{P}^i = \{\phi^i_l | 0 \leq k < \Phi^i\}$ the ordered set of phases (i.e. the multi-phase *profile*) representing the execution of task τ^i, with Φ^i the number of phases. Each ϕ^i_l is defined by:

- $\phi^i_l.d$: its start date.
- $\phi^i_l.dur$: its worst-case duration in isolation (without interference).
- $\phi^i_l.m$: the worst-case number of memory accesses that may be performed within $[\phi^i_l.d, \phi^i_l.d + \phi^i_l.dur[$.

The date of ϕ^i_0, which is also the start date of task τ^i without interference, is set when the static schedule of the system is built. Then, for each ϕ^i_l ($l > 0$) the start date is defined by:

$$\phi^i_l.d = \phi^i_0.d + \sum_{0 \leq q < l} \phi^i_q.dur = \phi^i_{l-1}.d + \phi^i_{l-1}.dur \tag{3.1}$$

In order to compute the worst-case number of memory accesses performed during a given phase (i.e. $\phi^i_l.m$), the code portions of τ^i that may be executed during ϕ^i_l must be identified and analyzed. To do so, we introduce $\mathbb{T}^i = \{t^i_j | 0 \leq j < T^i\}$ the set of *execution traces* of τ^i, where T^i is the number of traces. Each trace corresponds to a possible execution of τ^i (corresponding to a particular set of

[1] In modern processors, the actual accesses may not be performed as soon as the corresponding instruction is executed e.g. if a store buffer delays store operations. However it is possible to statically bound the time window during which the access can be performed. For clarity reasons, we consider in this manuscript only the date when the instruction initiates the access in the pipeline, but the model can be easily extended to work with an interval of potential access dates.

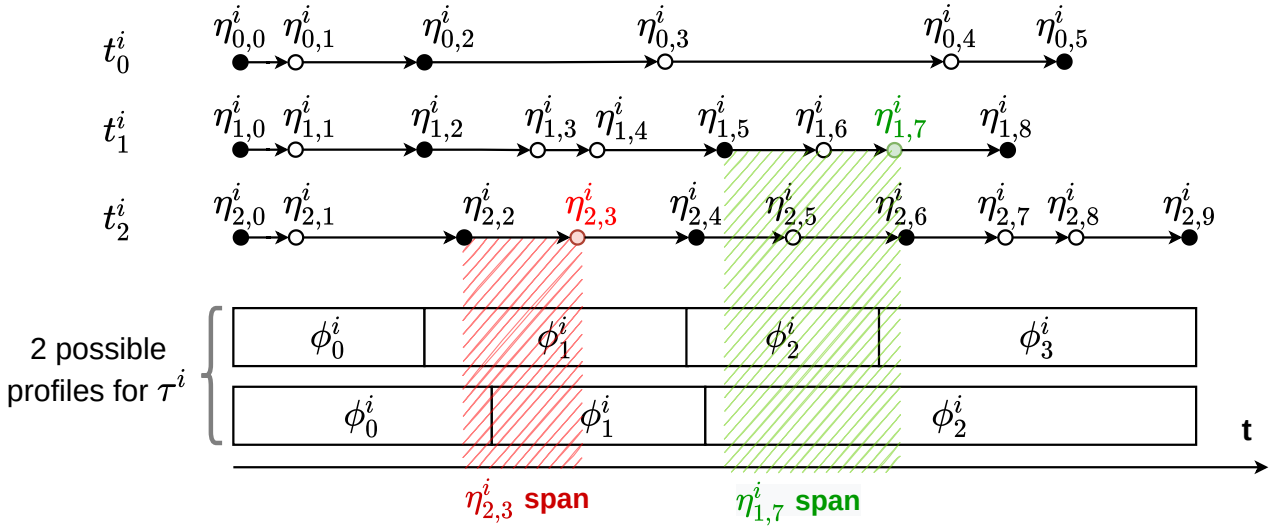


Figure 3.4: Three traces and two profiles for task τ^i. Red and green rectangles show the potential span of nodes $\eta_{2,3}^i$ and $\eta_{1,7}^i$ respectively.

inputs) and is a sequence of nodes $\eta_{j,k}^i$ representing instructions with $0 \leq k < N_j^i$ the node's index in its sequence. $\eta_{j,0}^i$ is the *entry point* of task τ^i and each node is defined by:

- $\eta_{j,k}^i.it$: the instruction represented by $\eta_{j,k}^i$. Here, an instruction is not just understood as an element of the core ISA (e.g. the ADD instruction), but as a particular instruction in the binary code of the task. Thus, nodes from different traces may reference the same instruction in the code.
- $\eta_{j,k}^i.m \in \mathbb{N}$: the worst-case number of memory accesses performed by $\eta_{j,k}^i.it$.
- $\eta_{j,k}^i.d$: the worst-case execution date of $\eta_{j,k}^i.it$ in trace t_j^i.

3.3.2 Synchronized nodes

As pointed out in Section 3.1.1, working on the complete set of execution traces of all tasks composing the system is not realistic. As a consequence, we formulate our correctness criteria using memory-centric *abstract traces*: the nodes composing the traces that we consider only represent the instructions that may perform memory accesses. The rest of the instructions is abstracted by computing local WCETs between consecutive memory accesses and accounting for these durations in the worst-case execution date of the nodes ($\eta_{j,k}^i.d$)[2]. As a result, in this model each node is guaranteed not to execute after its worst-case date, but is *a priori* able to execute anytime before this date. In order to account safely for the accesses in the phases, we thus would have to account for the accesses performed by a node in all phases that start before the worst-case date of this node. This would lead to huge over-approximations. In order to limit this approximation, some selected nodes must be *synchronized*: synchronization code is inserted in the program to ensure that the synchronized nodes cannot be executed before their worst-case date. The synchronization code can be added by the programmer

[2]Our criteria are also valid for simple tasks for which obtaining and manipulating the exact timed execution traces is possible.

directly in the source code of the tasks, by the compiler as part of a low-level compilation pass, or during an automatic code re-engineering process to adapt legacy code to the multi-phase model. We attract the reader's attention to two particular aspects of the model described in Section 3.3.1: (i) the execution date for nodes that reference the same instruction in different traces and (ii) the modelling of instructions inside loops which may appear multiple times in the same trace at different dates. Both these aspects have to do with the way synchronizations are implemented in the tasks. When complex synchronization mechanisms are used (e.g. that are aware of the current execution trace or of the iteration count in the current loop), the same memory instruction in the code may be modelled as two (or more) nodes with different dates, which perfectly fits the model. If, however, the synchronization mechanism is unaware of the context, the worst-case execution date of nodes that reference the same instruction on separate traces must be the same. Since the model uses worst-case dates, the date chosen for all these nodes must be the maximum date amongst them. Additionally, without a context-aware mechanism, synchronizations inside loops become impossible to implement, so the model naturally fits this case. We voluntarily keep the model as general as possible and make no assumption on the implementation of the synchronization mechanisms in order to formulate correctness criteria that apply in all circumstances.

To keep track of the synchronized nodes, we add the boolean attribute $\eta^i_{j,k}.sync$ that is true if the node is synchronized and false otherwise.

Using these synchronizations, the accesses performed by any node must only be accounted for in the phases that: (1) finish after the last synchronization prior to the node **AND** (2) start before the worst-case date of the node.

This is illustrated in Figure 3.4, which depicts 3 execution traces (t^i_0, t^i_1 and t^i_2) and 2 possible profiles for a task τ^i. Synchronized nodes are depicted in black in the traces. The red (resp. green) rectangle shows the time window in which the accesses of node $\eta^i_{2,3}$ (resp. $\eta^i_{1,7}$) must be accounted for. In the first profile, the accesses of $\eta^i_{1,7}$ must be considered in phases ϕ^i_2 and ϕ^i_3, whereas in the second profile, they would only be considered in ϕ^i_2.

It is important to note that since $\eta^i_{j,k}.d$ is a worst-case date, if node $\eta^i_{j,k}$ is synchronized, then its execution date is exactly[3] $\eta^i_{j,k}.d$. We denote $s_{last}(\eta^i_{j,k})$ the last synchronized node before $\eta^i_{j,k}$ in trace t^i_j. By convention, we set $s_{last}(\eta^i_{j,k}) = \eta^i_{j,k}$ when $\eta^i_{j,k}.sync$.

To account for the tasks schedule, for all tasks τ^i, the entry node (on any trace t^i_j) is synchronized and its worst-case execution date is set to the start of the first phase of the profile:

Property 2

$$\forall i, j : \eta^i_{j,0}.sync \wedge \eta^i_{j,0}.d = \phi^i_0.d$$

The worst-case date of any other node $\eta^i_{j,k}$ with $k > 0$ is defined according to the date of the last synchronized node on its trace:

Property 3: Worst-case execution date of node in isolation

$$\eta^i_{j,k}.d = \eta^i_{j,s}.d + \sum_{s \leq t < k} wcet(\eta^i_{j,t}.it, \eta^i_{j,t+1}.it)$$

where $wcet(\eta^i_{j,t}.it, \eta^i_{j,t+1}.it)$ is the WCET between instructions $\eta^i_{j,t}.it$ and $\eta^i_{j,t+1}.it$, and $\eta^i_{j,s}$ is $s_{last}(\eta^i_{j,k})$ if $\neg\eta^i_{j,k}.sync$ and $s_{last}(\eta^i_{j,k-1})$ otherwise.

[3] With a precision of a few cycles depending on the implementation of the synchronization mechanism.

A node $\eta^i_{j,k}$ can only be executed in the interval $[s_{last}(\eta^i_{j,k}).d, \eta^i_{j,k}.d]$. As we saw in the example of Figure 3.4, this interval may overlap with several phases of the task profile.

Definition 3: Restriction of trace t^i_j to phase ϕ^i_l

We denote $t^i_j|_{\phi^i_l}$ the set of nodes in trace t^i_j that may be executed within $[\phi^i_l.d, \phi^i_l.d + \phi^i_l.dur[$, called the *restriction* of trace t^i_j to phase ϕ^i_l:

$$t^i_j|_{\phi^i_l} = \{\eta^i_{j,k} | (\eta^i_{j,k}.d \geq \phi^i_l.d) \wedge (s_{last}(\eta^i_{j,k}).d < \phi^i_l.d + \phi^i_l.dur)\}$$

The notion of restriction of a trace to a phase is illustrated in Figure 3.5 on 3 traces over phase ϕ^i_1.

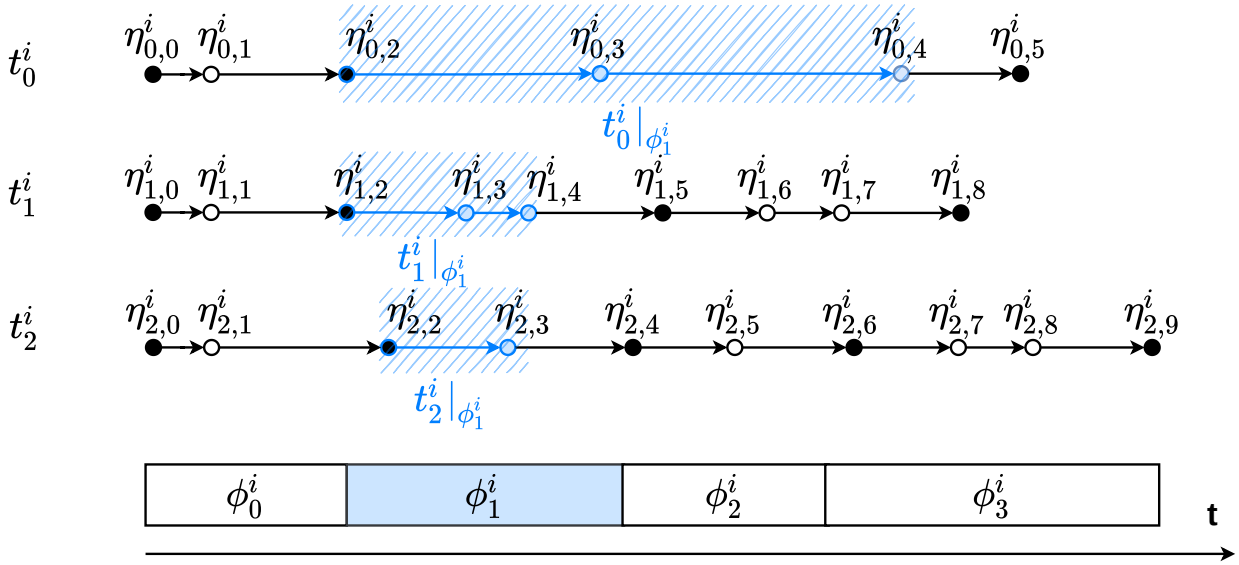


Figure 3.5: Restrictions of traces t^i_0, t^i_1 and t^i_2 to phase ϕ^i_1.

3.3.3 Maximum number of accesses in a phase

The number of accesses that may be performed during a phase for an individual trace is equal to the sum of the accesses of the nodes from this trace that may be executed in the phase. During the execution of a task, only one trace executes (which one depends on the execution context): as a consequence, the worst-case number of accesses performed during a phase is equal to the maximum number of accesses that may be performed by any execution trace during that phase.

Property 4: Number of accesses in a phase

The worst-case number of accesses that may be performed during phase ϕ_l^i, denoted $\phi_l^i.m$, is equal to the maximum of accesses per trace during phase ϕ_l^i:

$$\phi_l^i.m = \max_{0 \leq j < T^i} \Big(\sum_{\eta_{j,k}^i \in t_j^i |_{\phi_l^i}} \eta_{j,k}^i.m \Big)$$

Correctness criterion 1

The formula of Property 4 provides a conservative estimation of the number of memory accesses that can occur during the phases of a multi-phase profile.

Since nodes may span over multiple phases, the number of accesses counted task-wise may be overestimated, even when some nodes are synchronized. However, nodes from a trace which span over multiple phases may be "covered" by other nodes from another trace performing more accesses on a given phase. For example, in Figure 3.5, if we consider that each node performs 1 access, trace t_2^i is the local worst trace on ϕ_3^i with 4 nodes performing accesses and trace t_1^i is the local worst trace on ϕ_2^i with 3 nodes performing accesses. On phase ϕ_1^i, traces t_0^i and t_1^i both have 3 nodes performing accesses. In such circumstances, although node $\eta_{0,4}^i$ spans over ϕ_3^i, ϕ_2^i and ϕ_1^i, it does not contribute to any over-approximation.

We quantify the task-wise over-approximation of memory accesses compared to the 1-phase model, by computing the difference between the sum of accesses accounted for in each phase, and the worst trace-wise number of accesses.

Property 5: Over-approximation of memory accesses

The memory access over-approximation in a multi-phase profile of a task τ^i compared to its 1-phase representation is equal to:

$$\Delta = \Big(\sum_{0 \leq l < \Phi^i} \phi_l^i.m \Big) - \max_{0 \leq j < T^i} \Big(\sum_{0 \leq k < N_j^i} \eta_{j,k}^i.m \Big)$$

3.3.4 Multi-phase model and interference analysis

Notation	Definition
$\phi_l^i.p$	timing penalty added to ϕ_l^i due to potential interference
$\phi_l^i.d^{\#}$	*post-analysis* start date of ϕ_l^i
$\eta_{j,k}^i.d^{\#}$	worst-case date of node $\eta_{j,k}^i$ in the presence of interference

In this section, we consider a task system for which an analysis has provided a multi-phase model as well as a selection of synchronized nodes for each task. We assume that this task system is scheduled statically (the $\phi_0^i.d$ for each τ^i are selected and the start dates of the other phases are computed using Equation 3.1), and that an interference analysis (such as e.g. [26]) is applied to compute and account for the effect of potential interference between the tasks phases, assuming the timing-compositionality of the target processor [39]. In practice, each phase that potentially suffers from interference is extended

using a time penalty, and the next phases are postponed accordingly. This extension may violate assumptions that were made on the correspondence between phases and traces: in particular the restrictions of traces to phases that were computed prior to the interference analysis may no longer be correct, resulting in the possibility that some contentions between cores may happen in phases in which they were not accounted for.

3.3.4.1 Interference analysis hazards w.r.t. the multi-phase model

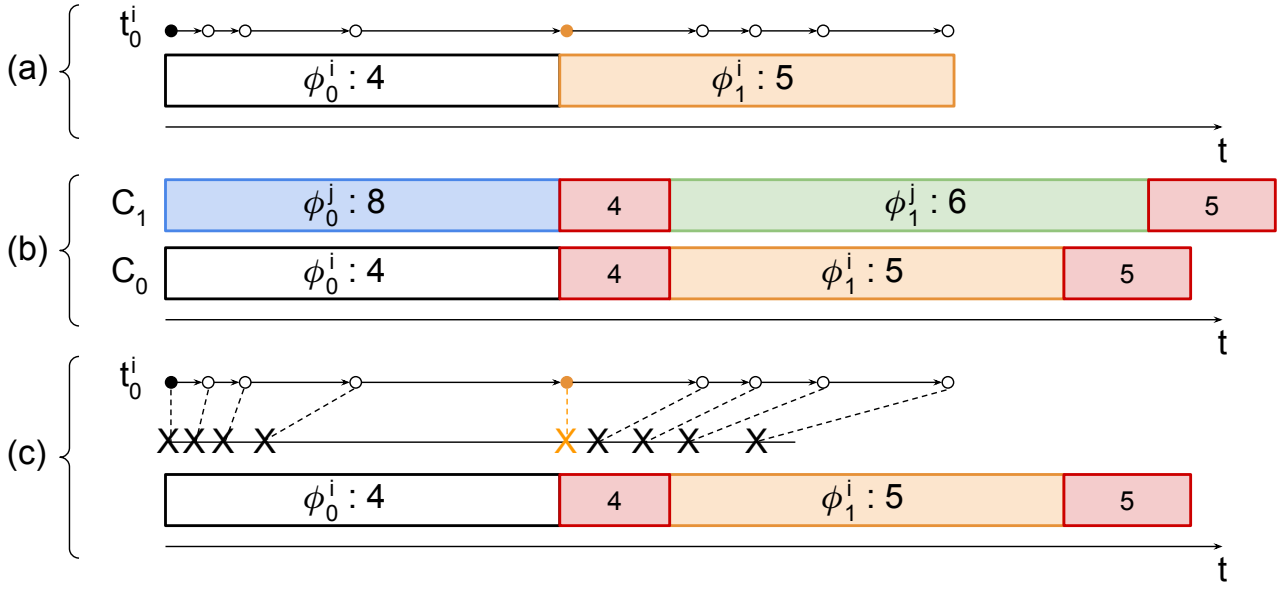


Figure 3.6: An example of incoherence that can appear between the multi-phase model and the actual execution of the code it represents.

We introduce the problem with the example of Figure 3.6. We assume that the analysis of a task τ^i results in the multi-phase model of (a): a first phase with 4 accesses in the worst case, followed by a second phase performing 5 accesses in the worst case. For the sake of simplicity, we also assume that this task only has one abstract execution trace t_0^i, also displayed in (a). The first access of each phase is synchronized to its worst-case date. Now, in (b), we assume that task τ^i is scheduled on core C_0, in parallel with task τ^j on core C_1. After performing an interference analysis, timing penalties (depicted in red) are added to each phase to account for the potential worst-case contentions. In (c), we display again the abstract execution trace t_0^i, in front of the interference-aware multi-phase representation of τ^i obtained in (b). We also display an actual execution trace of τ^i, in which the actual dates of the accesses are marked by crosses. We notice that three accesses from the second phase are actually performed at dates that now correspond to the extended first phase. However, these accesses are unaccounted for in the first phase, which jeopardizes the soundness of the analysis w.r.t. the actual execution.

In order to solve the problem, one option could be to update the count of the accesses in the interference-aware version of the multi-phase representation of each task, and to perform another interference analysis on the updated phases. This procedure should then be repeated until a fixed point is reached. This solution considerably complexifies the analysis, makes the resulting multi-phase profiles closer to their single-phase counterparts by extending the first phase at the expense of the others, and may add over-approximation to the access count, as the synchronizations may no longer

correspond to the profiles shapes. We instead chose to apply a simpler solution: postponing the execution date of synchronized nodes to the interference-aware worst-case start date of the phase they belong to.

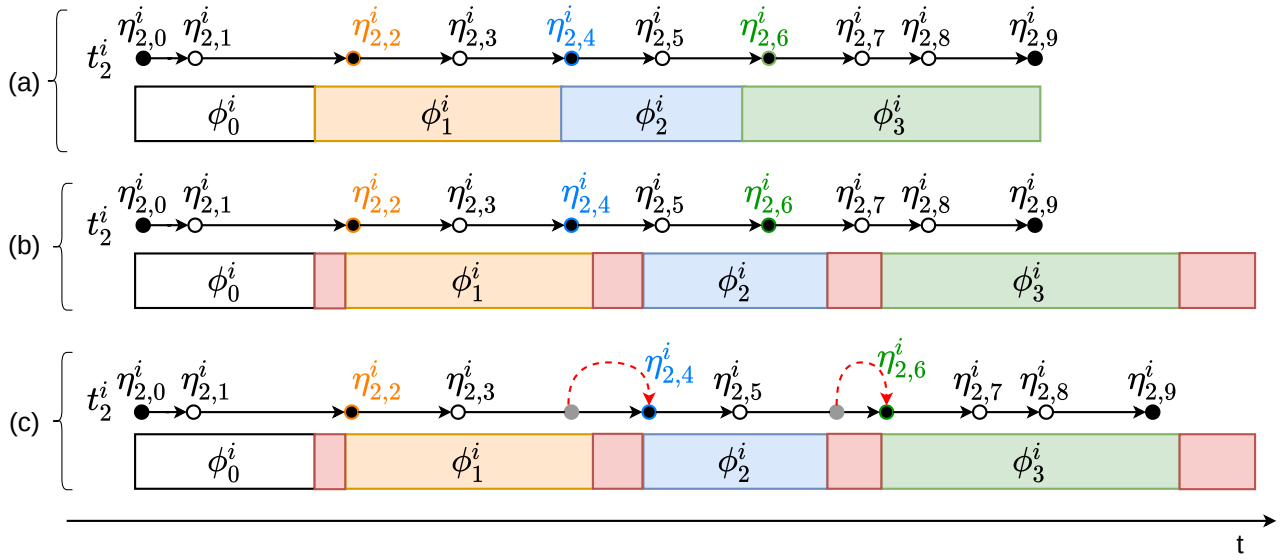


Figure 3.7: A trace and its corresponding phases representation : (a) in isolation, (b) after the interference analysis, red rectangles are the timing penalty added for each phase, (c) after a correction on nodes dates.

We illustrate this solution in the following example. Figure 3.7 displays trace t_2^i and the profile from Figure 3.5, at three stages of the analysis:

- (a) depicts the trace and phases before the interference analysis. We have:
  $t_2^i|_{\phi_0^i} = \{\eta_{2,0}^i, \eta_{2,1}^i\}$; $t_2^i|_{\phi_1^i} = \{\eta_{2,2}^i, \eta_{2,3}^i\}$; $t_2^i|_{\phi_2^i} = \{\eta_{2,4}^i, \eta_{2,5}^i\}$; $t_2^i|_{\phi_3^i} = \{\eta_{2,6}^i, \eta_{2,7}^i, \eta_{2,8}^i, \eta_{2,9}^i\}$

- (b) shows the same trace and profile after the interference analysis (assuming other tasks in the system): the effect of interference is materialized by timing penalties on the phases (the red rectangles after each phase). $t_2^i|_{\phi_1^i}$, $t_2^i|_{\phi_2^i}$ and $t_2^i|_{\phi_3^i}$ are different than in (a):
  $t_2^i|_{\phi_0^i} = \{\eta_{2,0}^i, \eta_{2,1}^i\}$; $t_2^i|_{\phi_1^i} = \{\eta_{2,2}^i, \eta_{2,3}^i, \eta_{2,4}^i, \eta_{2,5}^i\}$; $t_2^i|_{\phi_2^i} = \{\eta_{2,5}^i, \eta_{2,6}^i, \eta_{2,7}^i, \eta_{2,8}^i\}$; $t_2^i|_{\phi_3^i} = \{\eta_{2,8}^i, \eta_{2,9}^i\}$
  As a consequence, the worst-case amount of accesses that can happen during phases ϕ_1^i and ϕ_2^i is higher than what was assumed and therefore their interference penalty and those of the tasks scheduled in parallel are no longer conservative.

- (c) represents our solution to respect the model's assumptions of (a): the synchronized date of $\eta_{2,4}^i$ (resp. $\eta_{2,6}^i$) is set to the new starting date of ϕ_2^i (resp. ϕ_3^i), which is the unique phase in which it was accounted for in (a). With this slight modification, the restrictions of t_2^i to each phase are identical to the ones in (a) and the $\phi_l^i.m$ that was computed in isolation for each phase remains correct.

3.3.4.2 Enforcing the model's assumptions and the analysis results

Since the duration and start dates of phases can be changed as a result of the interference analysis, new attributes are added to the formal model of the phases:

- $\phi_l^i.p \geq 0$ is the timing penalty added to ϕ_l^i due to potential interference. It is a conservative bound computed during the interference analysis.
- $\phi_l^i.d^{\#}$ is the *post-analysis* date of ϕ_l^i, i.e. its start date taking into account the potential interference in the system.

After the interference analysis, the start date of some tasks may be postponed due to interference that delays previous tasks. $\phi_0^i.d^{\#}$ is thus fixed by applying the interference analysis results to the initial schedule. The start dates of all other phases ϕ_l^i describing the execution of τ^i are computed as:

$$\phi_l^i.d^{\#} = \phi_0^i.d^{\#} + \sum_{0 \leq q < l} (\phi_q^i.dur + \phi_q^i.p) = \phi_{l-1}^i.d^{\#} + \phi_{l-1}^i.dur + \phi_{l-1}^i.p \tag{3.2}$$

Correctness criterion 2

The synchronization dates in the final implementation of tasks must at least be equal to the start date of the corresponding phase: for each synchronization node $\eta_{j,k}^i \in t_j^i|_{\phi_n^i}$, the synchronization date is set to at least $\phi_n^i.d^{\#}$. This way it is guaranteed that nodes after $\eta_{j,k}^i$ cannot execute and thus produce accesses before the start of ϕ_n^i.

It seems straightforward that, by construction, a task set implemented using this rule is guaranteed to fulfill the assumptions made during the interference analysis: during the execution of the system, memory accesses will only occur at times that were accounted for during the analysis, and thus the amount of contentions cannot be larger in practice than what was accounted for. However, although this implementation rule directly guarantees that accesses are not performed before the phases in which they are accounted for, it may be harder to convince oneself that they cannot occur later than the end of these phases. Consequently, and given the potentially critical nature of the tasks modelled in the multi-phase representation, we provide in the remainder of the section a formal proof of the correctness of this implementation scheme w.r.t. the result of the interference analysis. Once again, this is completely agnostic of the analysis method, as long as it correctly provides a conservative bound on the interference level.

We denote $\eta_{j,k}^i.d^{\#}$ the *post-analysis* worst-case date of node $\eta_{j,k}^i$. The post-analysis dates of nodes are upper bounds on the worst-case execution dates of nodes in the presence of interference. We start by characterizing those bounds in our formal model (Properties 6, 7 and 8), and then use them to prove the correctness of the implementation of a multi-phase model of tasks.

First, the post-analysis execution date of the entry point of each task τ^i is the post analysis start date of its first phase ϕ_0^i.

Property 6

For any task τ^i: $\forall j < T^i, \eta_{j,0}^i.d^{\#} = \phi_0^i.d^{\#}$

Second, correctness criterion 2 has the following consequences for the post-analysis execution date of any synchronized node $\eta_{j,k}^i$ (except the entry point) of any task τ^i:

- if the phase ϕ_n^i in which the node was supposed to be executed is postponed due to interference penalties on previous phases, the node cannot be executed before the post-analysis start date of ϕ_n^i.

- if previous synchronized nodes see their execution dates postponed, the synchronization date of $\eta_{j,k}^i$ must be postponed accordingly, and thus computed from the post-analysis date of the previous synchronized node $\eta_{j,s}^i$. In this case, we must consider the interference that can take place between $\eta_{j,s}^i$ and $\eta_{j,k}^i$. If there exists one or more phases that span entirely between the two nodes, their penalties are added to the post-analysis date of $\eta_{j,k}^i$ (which is conservative). Moreover, by convention we count in the post-analysis date of $\eta_{j,k}^i$ the entire amount of penalty of the phase to which it belongs (which is also conservative since it accounts for the interference that can occur on each access in the phase prior to the synchronization node, and on each access that may occur until the next synchronization node).

Property 7

For any synchronized node $\eta_{j,k}^i$ of any trace t_j^i of task τ^i:

$$(k > 0 \wedge \eta_{j,k}^i.sync \wedge \eta_{j,k}^i \in t_j^i|_{\phi_n^i} \wedge \eta_{j,s}^i = s_{last}(\eta_{j,k-1}^i) \wedge \eta_{j,s}^i \in t_j^i|_{\phi_m^i})$$
$$\Rightarrow \eta_{j,k}^i.d^{\#} = max(\phi_n^i.d^{\#}, \eta_{j,s}^i.d^{\#} + \sum_{s \leq l < k} wcet(\eta_{j,l}^i.it, \eta_{j,l+1}^i.it) + \sum_{m < b \leq n} \phi_b^i.p)$$

Correctness criterion 3

The synchronization dates in the final implementation of tasks must not be set to a value higher than the date computed in Property 7.

Finally, for any non-synchronized node, its post-analysis date accounts for the possible postponing of the previous synchronized node $\eta_{j,s}^i$. Note that the potential interference occurring between them has been accounted for entirely in the post-analysis date of the previous synchronized node.

Property 8

For any non-synchronized node $\eta_{j,k}^i$ of any trace t_j^i of task τ^i:

$$(\neg\eta_{j,k}^i.sync \wedge \eta_{j,s}^i = s_{last}(\eta_{j,k}^i)) \Rightarrow \eta_{j,k}^i.d^{\#} = \eta_{j,s}^i.d^{\#} + \sum_{s \leq l < k} wcet(\eta_{j,l}^i.it, \eta_{j,l+1}^i.it)$$

3.3.4.3 Model correctness

We are now going to state the general correctness theorem: any task system that respects the 3 correctness criteria is correct w.r.t. the results of the interference analysis i.e. cannot generate interference that was not accounted for during the analysis.

The proof of this theorem relies on a lemma that states that the difference between the start date of a synchronized node $\eta_{j,k}^i$ before and after the interference analysis is bounded by the difference between the start date of the phase ϕ_l^i in which it is executed, before and after the interference analysis, added to the maximum effect of interference that can occur in ϕ_l^i. Next, we provide the lemma and the general theorem. The proofs are available in [56].

Lemma 3

$\forall \eta^i_{j,k}$: $(\eta^i_{j,k}.sync \wedge \eta^i_{j,k} \in t^i_j|_{\phi^i_l}) \Rightarrow \eta^i_{j,k}.d^{\#} - \eta^i_{j,k}.d \leq \phi^i_l.d^{\#} - \phi^i_l.d + \phi^i_l.p$

We now express the correctness property:

Theorem 15: Correctness of the multi-phase model w.r.t. its implementation

For any task system that respects correctness criteria 1, 2 and 3, for any $\eta^i_{j,k}$ of any task τ^i, if $\eta^i_{j,k}$ spans over a phase ϕ^i_l after the interference analysis, then $\eta^i_{j,k}$ was necessarily accounted in the restriction of trace t^i_j to ϕ^i_l before the analysis:
$\forall 0 \leq j < T^i, \forall 0 \leq k < N^i_j, \forall 0 \leq l < \Phi^i$:
$[s_{last}(\eta^i_{j,k}).d^{\#}, \eta^i_{j,k}.d^{\#}] \cap [\phi^i_l.d^{\#}, \phi^i_l.d^{\#} + \phi^i_l.dur + \phi^i_l.p[\neq \emptyset \Rightarrow \eta^i_{j,k} \in t^i_j|_{\phi^i_l}$

We just stated that the correctness criteria that we enumerated in this section guarantee that the implementation of a task system described in the multi-phase model is correct w.r.t. a chosen interference-aware static schedule. These criteria are very simple, which makes them easy to verify and offers a lot of room for optimizations in the analysis of tasks, both in order to derive a profile for tasks and to select the synchronization nodes.

In the next section, we describe the Time Interest Points framework that allows the derivation of multi-phase representations of tasks.

3.4 The Time Interest Points framework

In this section we first provide an overview of the TIPs static analysis framework, and then focus on each of the separate transformations that compose it.

Notation	Definition
φ^i_k	intermediate phase k for task τ^i
$\varphi^i_k.d$	start date of φ^i_k
$\varphi^i_k.dur$	worst-case duration of φ^i_k
$\varphi^i_k.end$	end date of φ^i_k
$\varphi^i_k.m$	map of traces to the maximum number of memory accesses they perform within ϕ^i_k
$[]$	empty sequence
$seq :: e$	concatenation of element e at the end of sequence seq
$\mathbf{0}$	constant function that maps 0 to any trace
$\mathbf{0}[t^i_j \longmapsto v]$	function that maps value v to t^i_j and 0 to any other trace

3.4.1 Overview of the Method and Models

The TIPs static analysis framework processes a real-time task system by a sequence of analyzes and transformations, which are detailed in the next sections:

- in a first step (Section 3.4.2), each task is analyzed in isolation. Starting from the disassembled binary of a task, a Control Flow Graph (CFG) is constructed. The CFG is analyzed in order

to extract TIPs, that is to say instructions that can produce or suffer from contentions. In our current implementations, we focus on instructions that may generate traffic on the memory bus due to a data cache miss, but the method could be easily extended to misses from instruction caches. Dealing with other potential sources of interference such as shared L2 caches or effects from cache coherence protocols will be addressed in future work.

- once the TIPs have been obtained, the CFG is transformed into a TIPsGraph (Section 3.4.2 as well): a simplified control flow graph where the nodes correspond to the TIPs of the task, and the edges represent the possible control flow between the TIPs, in an abstract version. Nodes are labelled with the number of memory accesses made by the corresponding TIP, and edges are labelled with the worst case execution time of any execution path linking the source TIP to the destination TIP of the edge. This representation is TIP-centered, and simplifies the CFG while allowing the following analyses and transformations to remain conservative.
- the TIPsGraph is then used to enumerate abstract execution traces using a working list algorithm (Section 3.4.3). The enumerated traces exhibit the occurrence of the TIPs in all possible executions of the task. For each trace, the TIPs execution dates are worst-case approximations.
- the traces for each task are then transformed into a sequence of phases (Section 3.4.4): each phase has a duration and a worst case number of memory accesses, and the sequence of phases represents an over-approximation of the number of memory accesses that can be performed by the task in the corresponding time windows.
- in the TIPs framework, the tasks of the system are then subjected to static scheduling, using their representation as sequences of phases (Section 3.5). During this step, an interference analysis is performed, which assumes that the processor architecture is time-compositionable [39], and its results are included in the schedule. Once an acceptable schedule (i.e. which respects all real-time constraints) has been found for the whole tasks system, synchronizations must be (automatically) inserted in the binary code of the tasks to enforce the schedule.

In the remainder of this section we provide more details and a formal representation for each of the aforementioned steps and models.

3.4.2 Extracting a TIPsGraph from a CFG

The analysis of each task τ^i in isolation starts by working on the CFG $CFG_{\tau^i} = \{\mathcal{N}, \mathcal{E}\}$ of τ^i, where $\mathcal{N}$ is the set of nodes called Basic Blocks (BBs) of the graph, and $\mathcal{E}$ is the set of edges $e \in \mathcal{N} \times \mathcal{N}$ that represent the control flow of the application. In this model BBs are sequences of instructions $i_0, i_1, ..., i_n \in \mathcal{I}$ with a single entry point and a single exit point. Using MUST and MAY cache analyses [53], TIPs are pinpointed from the rest of the instructions. As stated before, a TIP is an instruction that may create or suffer from interference. Remember that we focus on data cache misses as the sole source of interference in the system: a typical target would be a multi-core architecture in which each core has a private L1 data cache, a private scratchpad holding the code to execute and all cores share a first-come first-served memory bus. In this context TIPs are the memory instructions that cannot be statically determined to always result in a hit (called in short Always Hit - AH) in the L1 data cache of the core that executes them. The objective of the first step of the analysis is to build for each task τ^i a TIPsGraph $TG_{\tau^i} = \{\mathcal{T}, \mathcal{E}_{TG}\}$ where $\mathcal{T} \subseteq \mathcal{I} \times \mathbb{N}$ is the set of TIPs of the task and $\mathcal{E}_{TG} \subseteq \mathcal{T} \times \mathcal{T} \times \mathbb{N}$ is the set of edges representing the control flow between TIPs. Each TIP $tip \in \mathcal{T}$ is composed of an instruction $tip.it$ and of the worst case number of memory accesses $tip.m$ that this instruction may perform. Each edge $e \in \mathcal{E}_{TG}$ is composed of a couple of TIPs $(e.src, e.dst)$, as well

as a conservative approximation of the worst case execution time ($e.w$) of the code portions between $e.src.it$ and $e.dst.it$.

Property 9

$\forall e \in \mathcal{E}_{TG}$, $e = (tip_j, tip_k, e.w)$, $\forall p \in PATHS(tip_j, tip_k), e.w \geq WCET(p)$

where $PATHS(tip_j, tip_k)$ is the set of possible execution paths between instructions $tip_j.it$ and $tip_k.it$, and $WCET(p)$ is a conservative approximation of the WCET of the code portion composed of the instructions of p, which can be computed using a static analysis tool.

To ensure that a TIPsGraph covers the possible executions of the whole task it represents, we add two fictive nodes it_{entry} and it_{exit} that represent the entry and exit points of the task. Both $it_{entry}.m$ and $it_{exit}.m$ are equal to 0. Fig. 3.8a shows a TIPsGraph along with the CFG from which it was extracted. The TIPsGraph starts with node it_{entry} and ends with node it_{exit}. The rest of the nodes composing the TIPsGraph is extracted from the CFG: in this example we assume that four memory instructions may access the bus (the cache analysis did not result in AH for these). Each of them is represented in the TIPsGraph, as well as the possible control flow between them. Each edge records such a possible transition, and is labelled with the WCET of the portions of code that are executed between the TIP instructions.

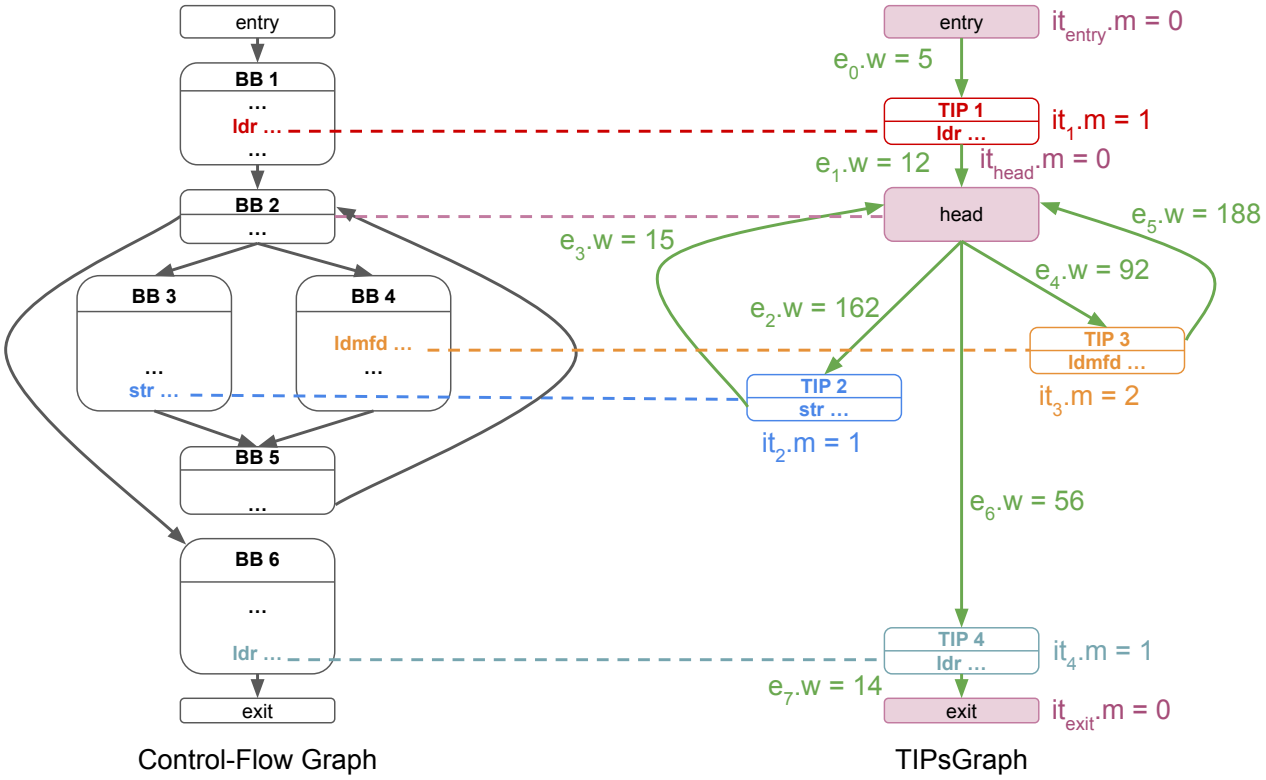
In order to correctly handle loops, a TIP it_{head}, with $it_{head}.m = 0$ is also created to represent the loop header BB, if and only if there exists at least a TIP tip inside the loop with $tip.m > 0$. When there is no TIP inside the loop, the loop gets abstracted in the TIPsGraph, like illustrated in Fig. 3.8b: the control flow of the loop is no longer detailed in the TIPsGraph, but the edge representing the transition between the last TIP before the loop and the first TIP after the loop accounts for the worst case loop duration.

3.4.3 Enumeration of Timed Execution Traces

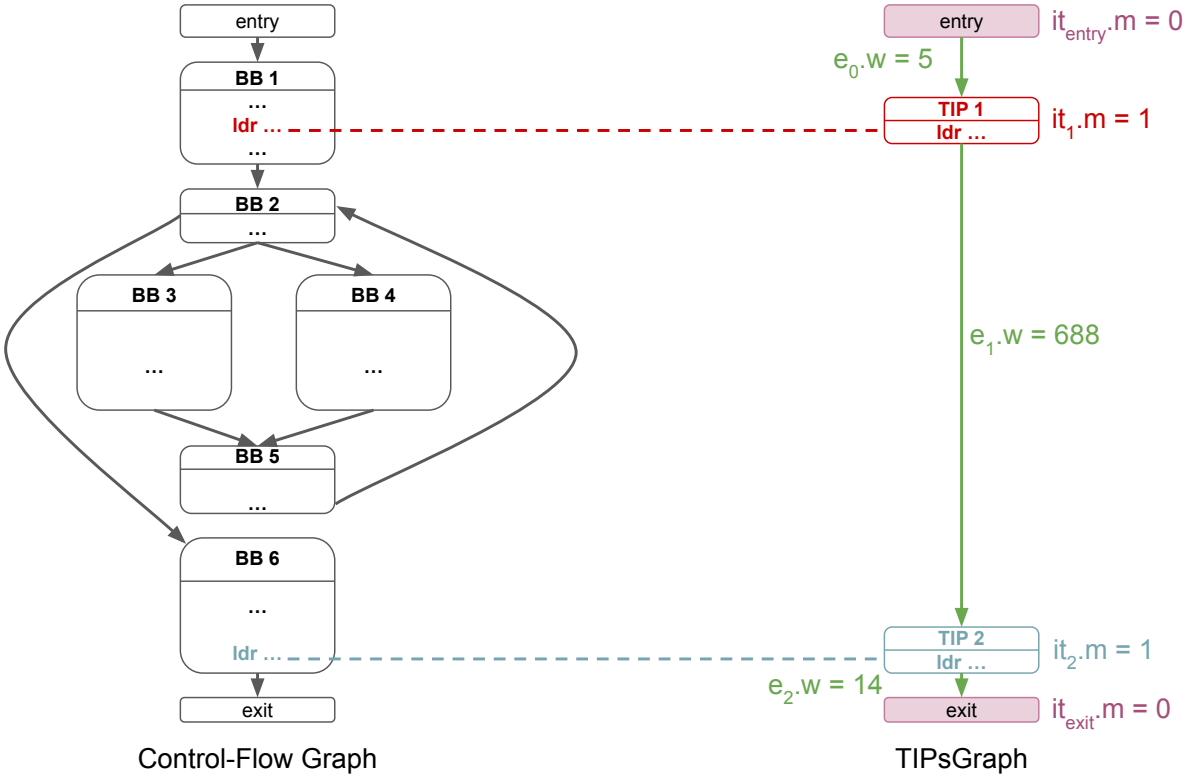
The next step of the analysis is to enumerate execution traces from the TIPsGraph. The result of the enumeration of the TIPsGraph TG_{τ^i} is the set of abstract execution traces $\mathbb{T}^i$, that we introduced in Section 3.3.1. In the description of our algorithm for building traces by enumeration of the paths of a TIPsGraph, we denote by $t^i_j :: \eta^i_{j,k}$ the extension of trace t^i_j with node $\eta^i_{j,k}$.

A basic enumeration algorithm is described in Algorithm 1. It is a working list algorithm that performs a depth-first traversal of the TIPsGraph of a task τ^i. The working list WL contains triplets composed of a trace currently under construction, a TIPsGraph edge and a stack containing information regarding the current iteration of loops that are being traversed. The algorithm iteratively builds the set $\mathbb{T}^i$ of the enumerated traces. Initially, WL and $\mathbb{T}^i$ are empty. The execution date of the first node of each trace, corresponding to it_{entry}, is arbitrarily set to 0. This way the dates of the other nodes in the traces are relative to the start date of the task. At each step of the process, the algorithm gets a trace under construction from WL, along with an edge from the TIPsGraph whose source node is the current last node of the trace, and the corresponding loop iteration context. From this, the trace is extended with the destination instruction of the edge. Then, for each possible successor edge e of the new last node of the trace, a copy of the current trace is created and pushed on WL with e and the current state of the context. One trace is completed and thus added to the $\mathbb{T}^i$ set when the node it_{exit} has been reached.

The tricky cases concern loop headers (**L.16** to **L.33**): in order for the algorithm to finish, it is mandatory for the number of iterations of each loop of the task to be bounded (which is a basic requirement for WCET computation). When the trace enumeration reaches an edge whose destination



(a) With TIPs in the loop



(b) Without TIPs in the loop

Figure 3.8: Example of CFGs and their corresponding TIPsGraphs

Algorithm 1 Basic trace enumeration

1: $\mathbb{T}^i \leftarrow \emptyset$
2: $j \leftarrow 0$
3: $WL = []$
4: **for all** $e \in \mathcal{E}_{TG}$ **s.t.** $e.src == i_{start}$ **do** ▷ *Initialize one or more traces*
5: $\quad \eta^i_{j,0}.it = i_{start}.it$
6: $\quad \eta^i_{j,0}.m = i_{start}.m$
7: $\quad \eta^i_{j,0}.d = 0$
8: $\quad t^i_j \leftarrow [\eta^i_{j,0}]$
9: $\quad push(WL, (t^i_j, e, []))$
10: $\quad j \leftarrow j + 1$
11: **end for**
12: **while** $WL \neq []$ **do**
13: $\quad (t^i_k, e, context) \leftarrow pop(WL)$
14: $\quad \eta^i_{k,cur} \leftarrow last(t^i_k)$
15: ▷ *Dealing with loops*
16: $\quad iteration \leftarrow pop(context)$
17: $\quad$ **if** $is\_loop\_head(e.dst)$ **then**
18: $\quad\quad$ **if** $is\_return\_edge(e)$ **then**
19: $\quad\quad\quad$ **if** $iteration = max\_bound(loop(e))$ **then**
20: $\quad\quad\quad\quad$ **continue** ▷ *Not a valid trace: dump it*
21: $\quad\quad\quad$ **else**
22: $\quad\quad\quad\quad push(context, iteration + 1)$ ▷ *Advance iteration counter*
23: $\quad\quad\quad$ **end if**
24: $\quad\quad$ **else**
25: $\quad\quad\quad push(context, 0)$ ▷ *Entering a new loop*
26: $\quad\quad$ **end if**
27: $\quad$ **else**
28: $\quad\quad$ **if** $is\_loop\_exit(e)$ **then**
29: $\quad\quad\quad$ **if** $iteration < min\_bound(loop(e))$ **then**
30: $\quad\quad\quad\quad$ **continue** ▷ *Not a valid trace: dump it*
31: $\quad\quad\quad$ **end if**
32: $\quad\quad$ **end if**
33: $\quad$ **end if**
34: ▷ *Adding a new element to the trace*
35: $\quad \eta^i_{k,cur+1}.it = e.dst.it$
36: $\quad \eta^i_{k,cur+1}.m = e.dst.m$
37: $\quad \eta^i_{k,cur+1}.d = \eta^i_{k,cur}.d + e.w$
38: $\quad t^i_k \leftarrow t^i_k :: \eta^i_{k,cur+1}$
39: $\quad$ **if** $e.dst = i_{end}$ **then**
40: $\quad\quad \mathbb{T}^i \leftarrow \mathbb{T}^i \cup \{t^i_k\}$
41: $\quad$ **else**
42: $\quad\quad$ **for all** $e_n \in \mathcal{E}_{TG}$ **s.t.** $e_n.src == e.dst$ **do**
43: $\quad\quad\quad t^i_j \leftarrow copy(t^i_k, j)$
44: $\quad\quad\quad push(WL, (t^i_j, e_n, context))$
45: $\quad\quad\quad j \leftarrow j + 1$
46: $\quad\quad$ **end for**
47: $\quad$ **end if**
48: **end while**

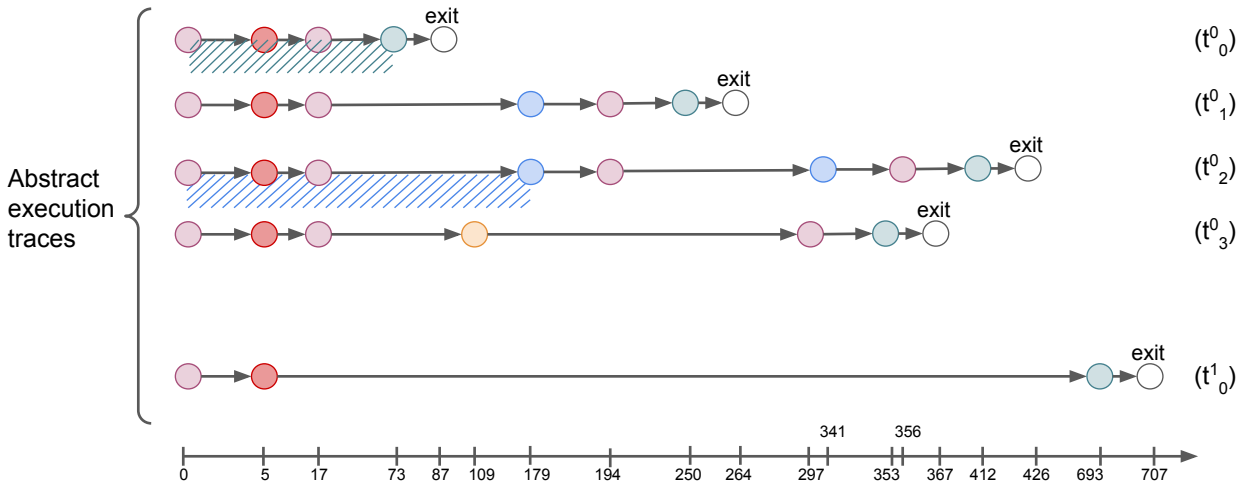


Figure 3.9: Examples of enumerated traces from the TIPsGraphs of Fig. 3.8

node corresponds to a loop header (**L.18**), the algorithm checks (**L.19**) whether the edge in question is a return edge from inside the loop (marking the end of an iteration of the loop), or not (meaning the enumeration is entering the loop for the first iteration). If the enumeration just enters the loop, a new loop iteration context is created by pushing 0 (corresponding to the first iteration of the loop) on the *context* stack (**L.25**). The algorithm uses a stack so it can handle nested loops. If on the other hand, the current edge is a return edge, the algorithm checks if the current iteration corresponds to a valid execution: it must not exceed the maximum iteration bound for the loop. If the execution is invalid, the current trace is simply discarded (**L.20**), and the algorithm pops a new element from WL. In order to work, the algorithm must also be able to pop an element from the context stack when exiting a loop. This is done by detecting that the current edge exits from the loop (**L.28**), and by checking that the minimum iteration bound has been reached in the current stack (**L.29**). This minimum iteration bound is set to 0 by default, but the more precise it is, the better the outcome of the analysis.

Figure 3.9 displays 5 traces enumerated from the TIPsGraphs of Fig. 3.8. The last trace (at the bottom), labelled (t^1_0) is the only trace that can be enumerated from the TIPsGraph of Fig. 3.8b. The first element of the trace, it_{entry}, corresponds to the start of the execution of the task at date 0. The next elements are the execution of $it_1.it$ at date 5, the execution of $it_4.it$ at date 693 and finally the end of the task at date 707. Traces (t^0_0) to (t^0_3) are a subset of all possible enumerated traces from the TIPsGraph of Fig.3.8a. In order to enumerate them, we assumed that the number of loop iterations varied at least between 0 iteration (trace (t^0_0)) and 2 iterations (trace (t^0_2)). Trace (t^0_0) corresponds to the execution of the task when the loop is not executed. Traces (t^0_1) and (t^0_2) correspond to the execution of the task when the left branch of the loop is taken respectively once and twice before exiting the loop. Trace (t^0_3) corresponds to the execution of the task when the right branch of the loop is taken once before exiting.

Once the traces of a task have been enumerated, the next step consists in generating its multi-phase profile.

3.4.4 Temporal phases

In this section, we present how a multi-phase profile is obtained for a task τ^i in the TIPs framework. The objective of the algorithms described in this section is to provide a profile for each task in isolation, prior to the scheduling phase. As a result, and until a scheduling pass is performed, each profile arbitrarily starts at date 0 (i.e. $\phi^i_0.d = 0, \forall i$).

The shape of the sequence of phases describing each task may impact the scheduling and interference analysis phase. A trade-off must be found between:

- the number of phases for each task. Scheduling elements (tasks, or phases) on a multi-core target is a NP-hard problem, so increasing the number of phases to schedule can increase the time it takes to build a schedule, potentially to a point where it is no longer feasible in practice.

- the length of the phases. During the interference analysis, any two phases from different tasks scheduled on overlapping time intervals on different cores are considered in interference. By definition, smaller phases occupy a core for less time than larger phases, and thus should be less exposed to interference from other cores. We wish to be able to tune the length of the phases in order to quantify its impact on the interference analysis.

- the worst case number of memory accesses in each phase. The length of the phases and the number and position of the synchronizations used to enforce them have an impact on the number of memory accesses attributed to each phase. This number must be conservative for each phase, so a memory access from a single instruction can be counted in multiple phases if the execution date of the instruction cannot be proven to happen in the time interval of only one phase. As discussed in Section 3.3 this can increase the over-estimation of memory accesses, or offer room to reduce the number of synchronizations to insert in the tasks.

- the number of necessary synchronizations to guarantee that the code corresponding to the phases does not start before the start date of the corresponding phase. One simple way to ensure that the code is correctly synchronized is to add one synchronization for the start of each phase on each trace. However, since each synchronization corresponds to additional code for the task, their number must remain limited. Optimizations can be used to reduce this number by e.g. removing the synchronizations that have no impact on the over-estimation of accesses.

In the remainder of this section, we provide algorithms that enable the extraction of valid multi-phase representations for tasks. These are baseline algorithms that do not perform any optimization with regard to the aforementioned trade-offs. In the description of the algorithms we use the empty sequence ($[]$) and concatenation of an element e at the right-end of a sequence seq ($seq :: e$).

Before obtaining task-level profiles such as the ones presented in Section 3.3, our algorithms create trace-wise intermediate phases that are then intersected and/or fused together. These intermediate phases have the same attributes as the final phases, but also record from which traces the accesses originate. This information allows to reduce the access over-estimation when phases are fused together. In order to avoid confusion between the final task-level phases and intermediate phases, we denote intermediate phases with the φ^i_l notation and keep the ϕ^i_m notation for the phases in the final profile. We add the $\varphi^i_l.end$ attribute to denote the end of φ^i_l ($\varphi^i_l.end = \varphi^i_l.d + \varphi^i_l.dur$). For intermediate phases, $\varphi^i_l.m$ is a map from the traces t^i_j of τ^i to the number of memory accesses that they initiate during φ^i_l.

Definition 4: Union operator on memory access maps

We define the union operator $\cup_+$ on two memory access maps $\varphi_l^i.m$ and $\varphi_n^i.m$:

$$\forall t_j^i, (\varphi_l^i.m \cup_+ \varphi_n^i.m)(t_j^i) = \varphi_l^i.m(t_j^i) + \varphi_n^i.m(t_j^i)$$

Our algorithms rely on the *Intersect* operator that is described in Definition 5. This operator computes the intersection of two intermediate phases: if the phases correspond to non-overlapping time intervals, the return value is empty. Otherwise, the operator returns an intermediate phase whose time interval is the intersection of the time intervals of the two input phases, and its worst case number of memory accesses map is the trace-wise union of the maps of the input phases.

Definition 5: Intersection function

Let $\varphi_l^i, \varphi_n^i,\ Intersect(\varphi_l^i, \varphi_n^i) =$

$$\begin{cases} \emptyset & if\ \varphi_l^i.d \geq \varphi_n^i.end \vee \varphi_n^i.d \geq \varphi_l^i.end \\ (\varphi_l^i.d, \varphi_l^i.dur, \varphi_l^i.m \cup_+ \varphi_n^i.m) & if\ \varphi_l^i.d \geq \varphi_n^i.d \wedge \varphi_n^i.end \geq \varphi_l^i.end \\ (\varphi_l^i.d, \varphi_n^i.end - \varphi_l^i.d, \varphi_l^i.m \cup_+ \varphi_l^i.m) & if\ \varphi_l^i.d \geq \varphi_n^i.d \wedge \varphi_l^i.end \geq \varphi_n^i.end \\ (\varphi_n^i.d, \varphi_l^i.end - \varphi_n^i.d, \varphi_l^i.m \cup_+ \varphi_n^i.m) & if\ \varphi_l^i.d < \varphi_n^i.d \wedge \varphi_l^i.end \leq \varphi_n^i.end \\ (\varphi_n^i.d, \varphi_n^i.dur, \varphi_l^i.m \cup_+ \varphi_n^i.m) & if\ \varphi_l^i.d < \varphi_n^i.d \wedge \varphi_l^i.end > \varphi_n^i.end \end{cases}$$

Our algorithms also use the *Trace_to_phases* procedure described in Algorithm 2. This procedure transforms a trace t_j^i of a task τ^i into a sequence of phases in the following manner: for each node $\eta_{j,k}^i$ in the trace, it creates two intermediate phases: φ_l^i that starts at the start date of the node, spans the worst case duration of the accesses of this node and has $\varphi_l^i.m(t_j^i) = \eta_{j,k}^i.m$ (marking that on this time interval trace t_j^i makes at most $\eta_{j,k}^i.m$ accesses), and φ_{l+1}^i that starts just after and spans until the date of the next node in the trace (i.e. $\eta_{j,k+1}^i$) and has $\varphi_{l+1}^i.m = \mathbf{0}$ (i.e. the constant function that maps all traces to 0). In case $\eta_{j,k}^i$ is the last node in trace t_j^i, φ_{l+1}^i spans until d_{max}, which is a value provided as a parameter to the procedure. The value of d_{max} should be the maximum of the dates of the last nodes of all traces of τ^i (i.e. the WCET of τ^i in the absence of interference).

The top-level procedure is described in Algorithm 3: starting with the first trace t_0^i from the set of traces of τ^i, it transforms it into a set of intermediate phases (*Inter_phases*) using procedure *Trace_to_phases*. Then each other trace $t_j^i \in \mathbb{T}^i$ is in turn transformed into a sequence of phases, and *Inter_phases* is updated with the intersection of these phases and the current phases of *Inter_phases*.

When this is done, a procedure reduces the number of phases using a minimum size δ, by :

- preserving all phases φ_j^i with $\varphi_j^i.m = \mathbf{0}$ and $\varphi_j^i.dur \geq \delta$,
- for all other phases, fusing consecutive phases until the result of the fusion has a length of at least δ or there is no more available phase to fuse. When fusing phases, the information about the worst case number of memory accesses is combined trace-wise instead of blindly summed in order to limit over-approximations, using the $\cup_+$ operator of Definition 4.

Finally, each intermediate phase φ_j^i is converted to its equivalent ϕ_k^i in the final task-level profile, by re-ordering them by increasing start date and setting $\phi_k^i.m$ to the maximum trace-wise number of

Algorithm 2 Trace_to_phases procedure

Require: $t^i_j \in \mathbb{T}^i$; $access\_time, d_{max} \in \mathbb{N}$
Ensure: $Inter\_phases$
1: $Inter\_phases = []$
2: $l \leftarrow 0$
3: **for all** $\eta^i_{j,k} \in t^i_j, k \neq N^i_j - 1$ **do**
4: $\quad \varphi^i_l.d \leftarrow \eta^i_{j,k}.d$ ▷ φ^i_l *accounts for the accesses*
5: $\quad \varphi^i_l.dur \leftarrow \eta^i_{j,k}.m \times access\_time$
6: $\quad \varphi^i_l.m \leftarrow \mathbf{0}[t^i_j \longmapsto \eta^i_{j,k}.m]$ ▷ $\varphi^i_l.m(t^i_j) = \eta^i_{j,k}.m$ *and 0 everywhere else*
7: $\quad \varphi^i_{l+1}.d \leftarrow \varphi^i_l.end$ ▷ φ^i_{l+1} *spans until the next access in the trace*
8: $\quad \varphi^i_{l+1}.dur \leftarrow \eta^i_{j,k+1}.d - \varphi^i_l.end$
9: $\quad \varphi^i_l.m \leftarrow \mathbf{0}$
10: $\quad Inter\_phases \leftarrow Inter\_phases :: \varphi^i_l :: \varphi^i_{l+1}$
11: $\quad l \leftarrow l + 2$
12: **end for**
13: $\varphi^i_l.d \leftarrow \eta^i_{j,N^i_j-1}.d$ ▷ *Last node in the trace*
14: $\varphi^i_l.dur \leftarrow \eta^i_{j,N^i_j-1}.m \times access\_time$
15: $\varphi^i_l.m \leftarrow \mathbf{0}[t^i_j \longmapsto \eta^i_{j,N^i_j-1}.m]$
16: $\varphi^i_{l+1}.d \leftarrow \varphi^i_l.end$ ▷ *Final, empty phase to span until* d_{max}
17: $\varphi^i_{l+1}.dur \leftarrow d_{max} - \varphi^i_l.end$
18: $\varphi^i_{l+1}.m \leftarrow \mathbf{0}$
19: $Inter\_phases \leftarrow Inter\_phases :: \varphi^i_l :: \varphi^i_{l+1}$
20: **return** $Inter\_phases$

Algorithm 3 Creation of a multi-phase profile for task τ^i

Require: $\mathbb{T}^i$, $\delta \in \mathbb{N}$, $d_{max} \in \mathbb{N}$
Ensure: $\mathbb{P}^i$
1: $Inter\_phases = Trace\_to\_phases(t^i_0, d_{max})$
2: **for all** $t^i_j \in \mathbb{T}^i$, $t^i_j \neq t^i_0$ **do**
3: $\quad Inter\_phases \leftarrow Intersect(Phases_\tau, Trace\_to\_phases(t^i_j, d_{max}))$
4: **end for**
5: $Inter\_phases \leftarrow Fusion(Phases_\tau, \delta)$
6: $\mathbb{P}^i \leftarrow Convert\_to\_profile(Inter\_phases)$
7: **return** $\mathbb{P}^i$

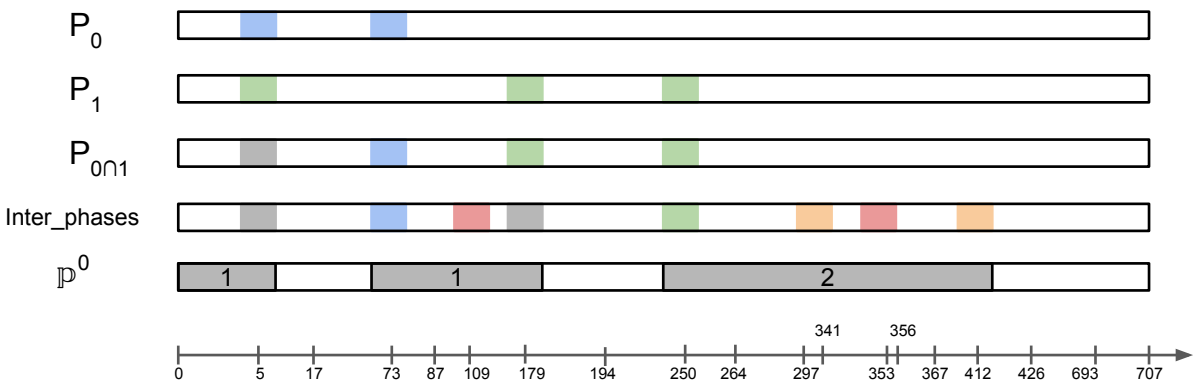


Figure 3.10: Examples of memory access profiles obtained from the traces of Fig. 3.9

accesses occurring in the phase: $max_l(\varphi^i_j.m(t^i_l))$, the rest of the attributes being directly copied from φ^i_j to ϕ^i_k.

We illustrate this algorithm in the examples of Figure 3.10. Intermediate phase sequences P_0 and P_1 are extracted respectively from traces t^0_0 and t^0_1 of Figure 3.9 using Algorithm 2. The result of their intersection is provided as $P_{0\cap1}$. In this sequence, the first access is displayed in gray to show that this phase corresponds to either one access from trace t^0_0 or one access from trace t^0_1. The sequence labelled *Inter_phases* is obtained by iterating the intersection of the traces t^0_0, t^0_1, t^0_2 and t^0_3. Different colors mean that accesses from different traces may occur. Finally the profile $\mathbb{P}^0$ is obtained by fusing together the smaller phases and preserving larger phases that are guaranteed not to perform any memory access.

This concludes our presentation of the TIPs framework. Now that we have presented a static analysis method to derive multi-phase profiles for tasks, we provide in the next section several algorithms that statically build schedules for multi-phase task systems, and try to take advantage of this representation to reduce the worst-case effect of interference.

3.5 Static scheduling of multi-phase tasks

This section discusses several approaches to benefit from the multi-phase representation when scheduling tasks on multi-core platforms. Our main objective is to minimize the makespan of the task system in the presence of interference.

3.5.1 Problem Definition

The following static scheduling problem is targeted: given a set of homogeneous cores[4] connected to a shared memory through a FCFS bus and a system composed of data-dependent tasks specified as a directed acyclic graph (DAG), schedule the tasks on the cores in order to minimize the interference-aware makespan of the system. This problem instance considers non-preemptive tasks only, and tasks are not partitioned to the cores prior to the scheduling phase.

[4]The proposed heuristics also apply to heterogeneous cores but experiments have not been conducted in this setting.

Notation	Definition
G	DAG defining task dependencies
E	set of edges (dependency relations) of G
$preds(\tau^i)$	set of predecessors of τ^i
$succs(\tau^i)$	set of successors of τ^i
$\mathbb{C}$	set of cores composing the architecture
C_k	core with index k
$\mathbb{S}$	schedule
$\mathbb{S}(C_k)$	schedule of core C_k
$\mathbb{S}(C_k).end$	end date of schedule of core C_k
$\phi^i_{\Phi^i}.d^\#$	end date of τ^i in the presence of interference
$\phi^i_j.\gamma$	number of potential contentions suffered by ϕ^i_j
$\phi^i_j.\gamma_k$	number of potential contentions suffered by ϕ^i_j from C_k
ω^i_k	True if τ^i is mapped to C_k
ρ^i_j	True if τ^i and τ^j are mapped to the same core
$\chi_{i,j\_k,l}$	True if the intervals covered by ϕ^i_j and ϕ^k_l overlap
$\theta_{i,j\_k,l}$	True if ϕ^i_j starts before the end of ϕ^k_l

Formally, let $\mathbb{C} = \{C_k | 0 \leq k < N_c\}$ be a multi-core architecture composed of N_c cores. Dependencies between the tasks of T are specified using a DAG $G = (T, E)$ in which vertices are the tasks of T and each edge $e_{i,j} \in E$ between τ^i and τ^j indicates that τ^i must be completed before τ^j can start. Moreover, $preds(\tau^i) = \{\tau^k | e_{k,i} \in E\}$ denotes the set of predecessors of τ^i and $succs(\tau^i) = \{\tau^k | e_{i,k} \in E\}$ the set of successors of τ^i.

Our objective is to build a schedule $\mathbb{S}$ of the tasks of T on the cores composing $\mathbb{C}$.

For each core C_k, the following attributes in $\mathbb{S}$ are defined:

- $\mathbb{S}(C_k)$: the schedule of C_k, which is a sequence of phases, ordered by their starting date.
- $\mathbb{S}(C_k).end$: the end date of the last phase scheduled on C_k.

The makespan of the task system in schedule $\mathbb{S}$ is $makespan(\mathbb{S}) = max_{C_k \in \mathbb{C}}(\mathbb{S}(C_k).end)$.

3.5.2 ILP Formulation

We now provide an ILP formulation of the problem. In this formulation we use bold font to denote the variables of the ILP system, ILP.1 is the objective function and the other equations numbered ILP.X are the constraints.

We first introduce variable $mksp$ denoting the makespan of the task system. It appears in the objective function that minimizes the makespan:

$$minimize\ \boldsymbol{mksp} \tag{ILP.1}$$

We use $\boldsymbol{\phi^i_{\Phi^i}.d^\#}$ to denote the end date of τ^i, which is the end date of its last phase:

$$\forall \tau^i, \boldsymbol{\phi^i_{\Phi^i}.d^\#} = \boldsymbol{\phi^i_{\Phi^i-1}.d^\#} + \phi^i_{\Phi^i-1}.dur + \boldsymbol{\phi^i_{\Phi^i-1}.p} \tag{ILP.2}$$

The makespan of the system is greater than the end date of all tasks:

$$\forall \tau^i, \boldsymbol{mksp} \geq \boldsymbol{\phi^i_{\Phi^i}.d^\#} \tag{ILP.3}$$

Moreover, each task τ^i starts after date 0 and after the end of all its predecessors:

$$\forall \tau^i, \boldsymbol{\phi_0^i.d^\#} \geq 0 \tag{ILP.4}$$

$$\forall \tau^k \in preds(\tau^i), \boldsymbol{\phi_0^i.d^\#} \geq \boldsymbol{\phi_{\Phi^k}^k.d^\#} \tag{ILP.5}$$

Following the definition of the start date of a phase in Equation 3.2, we can express the date of each subsequent phase as:

$$\forall \tau^i, \forall 0 < j \leq \Phi^i, \boldsymbol{\phi_j^i.d^\#} = \boldsymbol{\phi_{j-1}^i.d^\#} + \phi_{j-1}^i.dur + \boldsymbol{\phi_{j-1}^i.p} \tag{ILP.6}$$

We use boolean variable $\boldsymbol{\omega_k^i}$ to express the mapping of task τ^i: $\boldsymbol{\omega_k^i} = 1$ if and only if τ^i is mapped on C_k. Each task is mapped to a unique core so we add the constraints:

$$\forall \tau^i : \sum_{0 \leq k < N_c} \boldsymbol{\omega_k^i} = 1 \tag{ILP.7}$$

We also introduce variable $\boldsymbol{\rho_j^i}$ that is equal to 1 if and only if τ^i and τ^j are mapped to the same core:

$$\forall \tau^i, \tau^j, \boldsymbol{\rho_j^i} = \sum_{0 \leq k < N_c} \boldsymbol{\omega_k^i} \wedge \boldsymbol{\omega_k^j}$$

Because of the conjunction $\wedge$, the above equation is not linear. Therefore, we have to use a new variable $\boldsymbol{\Omega_k^{i,j}} = \boldsymbol{\omega_k^i} \wedge \boldsymbol{\omega_k^j}$ and add the following equations:

$\forall \tau^i, \tau^j, 0 \leq k < N_c$,

$$\boldsymbol{\Omega_k^{i,j}} \leq \boldsymbol{\omega_k^i} \tag{ILP.8}$$

$$\boldsymbol{\Omega_k^{i,j}} \leq \boldsymbol{\omega_k^j} \tag{ILP.9}$$

$$\boldsymbol{\Omega_k^{i,j}} + 1 \geq \boldsymbol{\omega_k^i} + \boldsymbol{\omega_k^j} \tag{ILP.10}$$

Therefore, the equation becomes:

$$\boldsymbol{\rho_j^i} = \sum_{0 \leq k < N_c} \boldsymbol{\Omega_k^{i,j}} \tag{ILP.11}$$

In the following, any other conjunction will be converted to a linear form in the same way. For clarity reasons, we do not provide the details for the other linearizations of conjunctions.

Two phases may contend with each other if they are scheduled on different cores and their execution intervals overlap. We introduce the boolean variable $\boldsymbol{\chi_{i,j\_k,l}}$ that is true if the intervals covered by ϕ_j^i and ϕ_l^k overlap:

$$\boldsymbol{\chi_{i,j\_k,l}} \Leftrightarrow \neg((\boldsymbol{\phi_{l+1}^k.d^\#} \leq \boldsymbol{\phi_j^i.d^\#}) \vee (\boldsymbol{\phi_{j+1}^i.d^\#} \leq \boldsymbol{\phi_l^k.d^\#}))$$
$$\boldsymbol{\chi_{i,j\_k,l}} \Leftrightarrow (\boldsymbol{\phi_j^i.d^\#} < \boldsymbol{\phi_{l+1}^k.d^\#}) \wedge (\boldsymbol{\phi_l^k.d^\#} < \boldsymbol{\phi_{j+1}^i.d^\#})$$

The overlapping is illustrated by Figure 3.11. Phase ϕ_0^k overlaps with ϕ_2^i but not with ϕ_0^j so $\chi_{k,0\_i,2} = \chi_{i,2\_k,0} = 1$ and $\chi_{k,0\_j,0} = \chi_{j,0\_k,0} = 0$.

We need to decompose the equivalence relation into several constraints in the ILP system. That is why we define $\boldsymbol{\theta_{i,j\_k,l}}$ as:

$$\boldsymbol{\theta_{i,j\_k,l}} \Leftrightarrow \boldsymbol{\phi_j^i.d^\#} < \boldsymbol{\phi_{l+1}^k.d^\#}$$

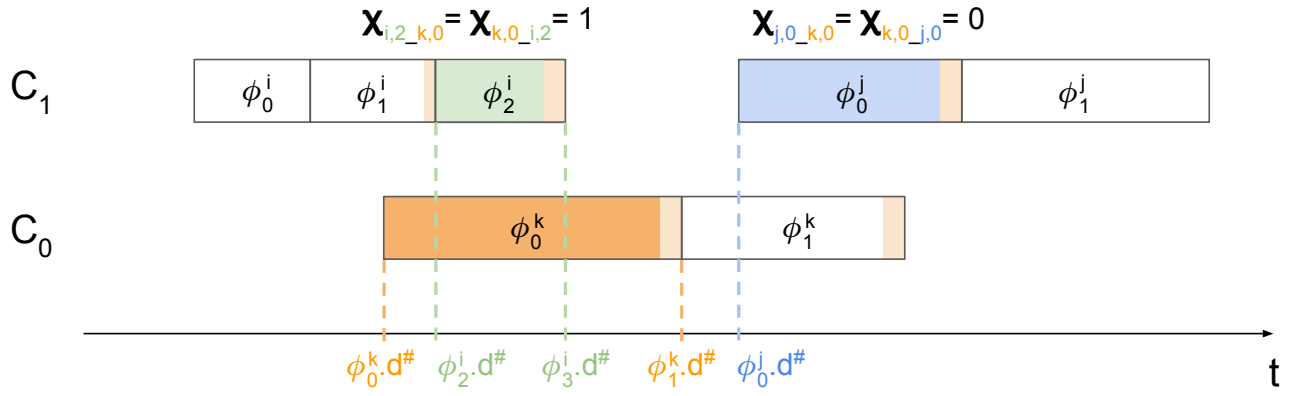


Figure 3.11: 3 tasks scheduled on 2 cores

so that the equivalence becomes:

$$\boldsymbol{\chi_{i,j\_k,l}} \Leftrightarrow \boldsymbol{\theta_{i,j\_k,l}} \wedge \boldsymbol{\theta_{k,l\_i,j}} \tag{ILP.12}$$

$\boldsymbol{\theta_{i,j\_k,l}}$ is defined using the big-M notation and a cancellation variable $\boldsymbol{\beta_{i,j\_k,l}}$:

$\forall \tau^i, \tau^j, 0 \leq j < \Phi^i, 0 \leq i < \Phi^k,$

$$1 + \boldsymbol{\phi_j^i.d^\#} \leq \boldsymbol{\phi_{l+1}^k.d^\#} + M(1 - \boldsymbol{\theta_{i,j\_k,l}}) \tag{ILP.13}$$

$$\boldsymbol{\phi_j^i.d^\#} \geq \boldsymbol{\phi_{l+1}^k.d^\#} - M(1 - \boldsymbol{\beta_{i,j\_k,l}}) \tag{ILP.14}$$

$$\boldsymbol{\beta_{i,j\_k,l}} + \boldsymbol{\theta_{i,j\_k,l}} = 1 \tag{ILP.15}$$

The overlapping of 2 phases is forbidden if their tasks (resp. τ^i and τ^k) are scheduled on the same core ($\boldsymbol{\rho_k^i} = 1$). Therefore:

$$\boldsymbol{\chi_{i,j\_k,l}} \leq 1 - \boldsymbol{\rho_k^i} \tag{ILP.16}$$

In order to compute the time penalty of ϕ_j^i, we multiply the number of contentions it may encounter ($\boldsymbol{\phi_j^i.\gamma}$) by the cost of one penalty denoted *penalty_cost*, so we have:

$$\forall \tau^i, 0 \leq j < \Phi^i, \boldsymbol{\phi_j^i.p} = \boldsymbol{\phi_j^i.\gamma} \times penalty\_cost \tag{ILP.17}$$

$\boldsymbol{\phi_j^i.\gamma}$ is the sum of the contentions that may be caused by tasks on all the cores:

$$\forall \tau^i, 0 \leq j < \Phi^i, \boldsymbol{\phi_j^i.\gamma} = \sum_{0 \leq k < N_c} \boldsymbol{\phi_j^i.\gamma_k} \tag{ILP.18}$$

with $\boldsymbol{\phi_j^i.\gamma_k}$ the number of contentions that ϕ_j^i may experience from tasks scheduled on core k. As we consider a shared memory bus following a FIFO policy, $\boldsymbol{\phi_j^i.\gamma_k}$ is bounded by $\phi_j^i.m$:

$$\forall \tau^i, 0 \leq j < \Phi^i, 0 \leq k < N_c, \boldsymbol{\phi_j^i.\gamma_k} = \min(\phi_j^i.m, \sum_{\tau^l \in T} \sum_{0 \leq q < \Phi^l} \phi_q^l.m \times (\boldsymbol{\chi_{i,j\_l,q}} \wedge \boldsymbol{\omega_k^l})) \tag{3.3}$$

The term $(\boldsymbol{\chi_{i,j\_l,q}} \wedge \boldsymbol{\omega_k^l})$ states that ϕ_j^i receives contentions from ϕ_q^l if and only if ϕ_q^l is mapped to core k and overlaps with ϕ_j^i.

Finally, to linearize the minimum operator, we use the following equations with $\boldsymbol{\alpha^i_{j,k}} \in \{0,1\}$ guaranteeing that one of the proposed values is taken:

$\forall \tau^i, \tau^j, 0 \leq j < \Phi^i, 0 \leq k < N_c,$

$$\boldsymbol{\phi^i_j}.\boldsymbol{\gamma_k} \leq \phi^i_j.m \tag{ILP.19}$$

$$\boldsymbol{\phi^i_j}.\boldsymbol{\gamma_k} \leq \sum_{\tau^l \in T} \sum_{0 \leq q < \Phi^l} \phi^i_j.m \times (\boldsymbol{\chi_{i,j\_l,q}} \wedge \boldsymbol{\omega^l_k}) \tag{ILP.20}$$

$$\boldsymbol{\phi^i_j}.\boldsymbol{\gamma_k} \geq (\sum_{\tau^l \in T} \sum_{0 \leq q < \Phi^l} \phi^i_j.m \times (\boldsymbol{\chi_{i,j\_l,q}} \wedge \boldsymbol{\omega^l_k})) - M \times \boldsymbol{\alpha^i_{j,k}} \tag{ILP.21}$$

$$\boldsymbol{\phi^i_j}.\boldsymbol{\gamma_k} \geq \phi^i_j.m - M(1 - \boldsymbol{\alpha^i_{j,k}}) \tag{ILP.22}$$

3.5.3 Heuristics

Computing the penalties directly in the optimization problem is inherently non-linear. As a consequence, we observe in practice that the ILP resolution time does not scale when the number of tasks or phases grows. In order to tackle large systems that cannot be handled by ILP, we designed heuristics. We present three of them in this section.

In the following algorithms, we use the *computeContentions*($\mathbb{S}$) function to compute the values of the $\phi^i_j.d^{\#}$ by applying the formulas from ILP.17 and Equation 3.3 on each phase ϕ^i_j of $\mathbb{S}$[5].

3.5.3.1 Greedy policies

The first scheduling policies that we present are two variants of list scheduling: the algorithm selects a task from a list of ready tasks, schedules it following the policy, updates the list of ready tasks, and iterates until all tasks have been scheduled.

As Soon As Possible scheduling (ASAP) The ASAP policy takes the current partial schedule (initially empty) and builds as many schedules as there are cores in $\mathbb{C}$ by selecting a task and scheduling it as soon as possible on each of the cores, without preemption and while respecting task dependencies. Once the ASAP date is determined for a core, all the phases of the task are scheduled according to Equation 3.1. It then selects the partial schedule that has the lowest makespan and moves on to the next task. The interference analysis is performed only once all the tasks of the system have been scheduled. Consequently, this is the simplest and the fastest algorithm of all the presented heuristics.

Starting Date Enumeration (SDE) The ASAP strategy is not always the best choice to minimize the makespan in the presence of interference. For instance, Figure 3.12 shows 3 different ways to schedule a new task (the purple one) on core C_1. At the top, when scheduling the task as soon as possible, the phase with 10 accesses overlaps with 2 other phases in parallel and creates in the worst case 13 (8+5) contentions on core C_0 (depicted in orange). In the schedule below, we postponed the task start date to the end of the phase with 8 accesses so the 10-accesses phase may only create 5 contentions, and this choice yields a reduction of the makespan. In the last schedule at the bottom,

[5]In our experiments, we use an efficient Python implementation of this function

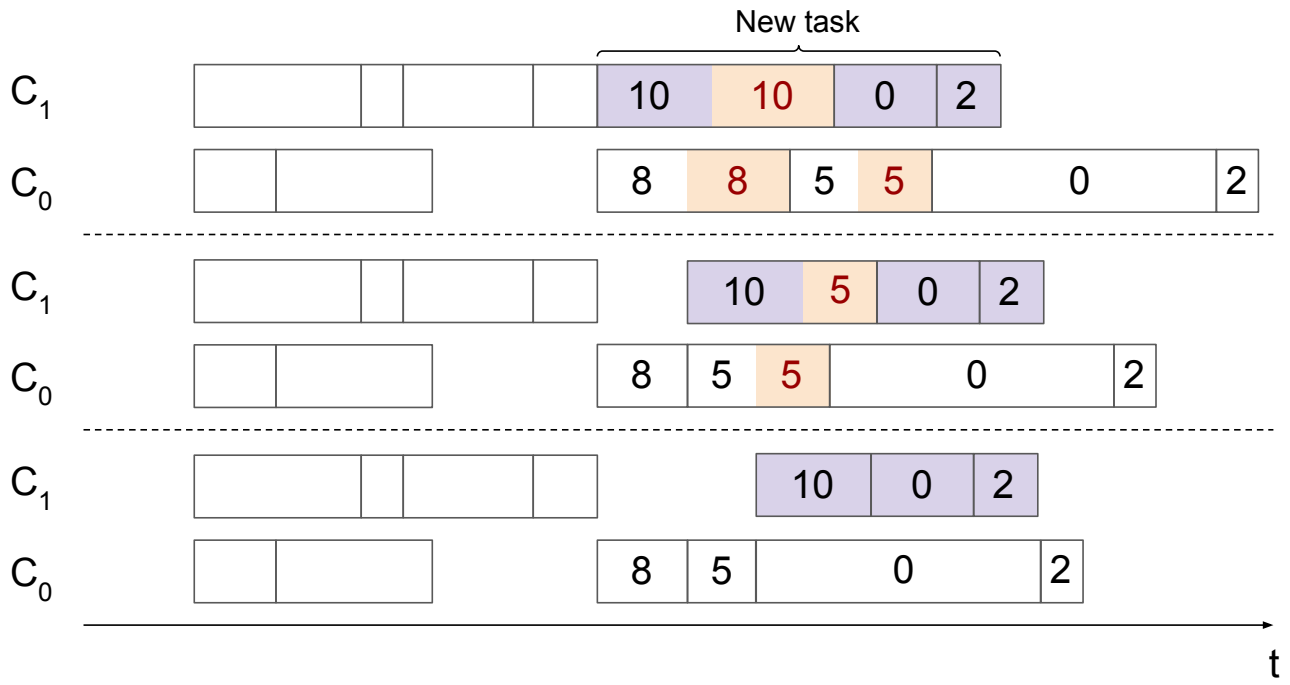


Figure 3.12: 3 different placements for a new task: the numbers within phases indicate their worst-case number of accesses and the orange rectangles are the additional penalty due to possible interference.

the task is postponed even more, to the next starting date of a phase in parallel. This results in no interference at all, yielding the smallest makespan. Following that idea, we developed the *SDE* heuristic that attempts to schedule the current task at several dates on each of the cores and performs an interference analysis for each possibility before selecting the one that minimizes the makespan.

Algorithm 4 describes SDE. It takes as inputs the current task to schedule, τ^i, and the current partial schedule $\mathbb{S}$, on which an interference analysis has been performed. The enumeration of the possible start dates for τ^i is limited to the interval $[\![minDate, maxDate]\!]$ in which minDate is the earliest possible start date of τ^i due to precedence constraints, and maxDate is the current makespan of the partial schedule. For each core C_k (line: 4), function *parallelDates* extracts the start and end dates of phases scheduled on the other cores that fall within the $[\![minDate, maxDate]\!]$ interval. Then, τ^i is iteratively scheduled at each of these dates on C_k in $\mathbb{S}$ (line: 7), and only the result yielding the smallest makespan (after interference analysis) is kept (line: 9). In the end, τ^i is scheduled on the core and at the date that yields the best makespan.

SDE considers candidate dates only for the start of the first phase of the task. The method could be extended to consider these dates as possible start dates for each of the phases, but the algorithmic complexity would increase accordingly, making the method impractical.

3.5.3.2 Iterative Priority Scheduling Heuristic (IPH)

The IPH, detailed in Algorithm 5, is an adaptation of the main algorithm of [45]. This algorithm has already been successfully adapted to the AER model in [55], but our task model is more generic and some assumptions made in [55] are not applicable here. As a consequence Algorithms 5 and 6 were adapted from [45] to handle the multi-phase model.

Algorithm 4 SDE

Require: τ^i ; $\mathbb{S}$
1: $minDate = max_{\tau^h \in preds(\tau^i)}(\phi^h_{\Phi^h}.d^{\#})$
2: $maxDate = makespan(\mathbb{S})$
3: $bestMakespan, bestSched = +\infty, \mathbb{S}$
4: **for** C_k **in** $\mathbb{C}$ **do**
5: $dates = parallelDates(\mathbb{S}, C_k, minDate, maxDate, \tau^i)$
6: **for** d **in** $dates$ **do**
7: $\mathbb{S}' = scheduleTask(\mathbb{S}, C_k, \tau^i, d)$
8: $computeContentions(\mathbb{S}')$
9: **if** $makespan(\mathbb{S}') < bestMakespan$ **then**
10: $bestMakespan = makespan(\mathbb{S}')$
11: $bestSched = \mathbb{S}'$
12: **end if**
13: **end for**
14: **end for**
15: **return** bestSched

The principle of this algorithm is to iteratively test different combinations of task priorities, called *priority vectors*, while converging to the best makespan, given by the objective variable *Obj* until no progress is made. In the initialization, we build the initial best schedule $\mathbb{S}^{best}$ using our ASAP greedy heuristic. Then, $\mathbb{S}^{best}$ is used to build the initial target interval for the makespan $[\![LB, UB]\!]$ (line 1) and *Obj* is chosen as the median value of this target interval. The initial values of the bounds do not have a huge impact on the quality of the result because the interval is re-adjusted throughout the iterations, but setting them close to a viable objective can save a few initial iterations. In order to speed up the computations, we implemented and when necessary, adapted, the following optimizations that were present in the original algorithm of [45]:

1. Using a symmetric instance of the scheduling problem because scheduling backwards may open scheduling options. That is why we distinguish the two graphs $G^{forward}$ and $G^{backwards}$ (line 5) and the priority vectors on both directions (line 34).

2. Implementing the algorithm in parallel so that several priority vectors are tested at the same time by separate threads.

3. Using a hash set to store the priority vectors that have already been tried to avoid repetitions (line: 6).

4. Modifying the priority vector using information about conflicting tasks that prevent each other to be scheduled before *Obj* (line: 33).

Optimizations 1 and 2 were directly implemented in our heuristic. We adapted optimization 3 to exploit the fact that two different priority vectors may produce the same scheduling order because of tasks dependencies. For example, if we consider tasks A, B and C with B and C successors of A, then assigning priorities 3, 2, 1 to respectively A, B and C yields the same scheduling order (A then B then C) as when assigning priorities 2, 3, 1 because task A must be executed before B and C has a lower priority than B. Therefore, instead of saving the priority vectors in the hash set, our algorithm computes and saves an equivalence class of the priority vectors given the dependencies of the system

Algorithm 5 IPH

Require: $G = (T, E), \mathbb{C}$
1: $UB, LB, \mathbb{S}^{best} = init(G, \mathbb{C})$
2: $Obj = (LB + UB)/2$
3: $failCount = 0$
4: $G^{forward} = G$
5: $G^{backward} = reverse(G)$
6: $prioHashSet = \{\}$
7: $init\_prio = [UB - \phi_0^i.d^{\#}]_{\forall \tau^i \in T}$
8: $sQueue = [(G^{forward}, Obj, init\_prio)]$
9: **while** $(LB < UB) \wedge (sQueue \neq [])$ **do**
10: $(G^c, Obj, prio) = sQueue.pop()$
11: $hash = Hash(eq\_class(prio, G^c))$
12: **if** $hash \in prioHashSet$ **then**
13: continue
14: **end if**
15: $prioHashSet.add(hash)$
16: $\mathbb{S} = findSchedule(G^c, \mathbb{C}, Obj, prio)$
17: **if** $makespan(\mathbb{S}) < makespan(\mathbb{S}^{best})$ **then**
18: $\mathbb{S}^{best} = \mathbb{S}$
19: **if** $UB > makespan(\mathbb{S})$ **then**
20: $UB, LB = update(UB, LB, \mathbb{S})$
21: **end if**
22: $Obj^{new} = UB - 100$
23: $priority = [Obj - \phi_0^i.d^{\#}]_{\forall \tau^i \in T}$
24: **else**
25: $failCount++$
26: **if** $failCount \geq log2(|T|)$ **then**
27: $LB = LB + (UB - LB)/4$
28: $failCount = 0$
29: **end if**
30: $Obj^{new} = \lceil min(UB, 1.1 \times Obj) \rceil$
31: **end if**
32: $prio_1 = [Obj - prio[i]]_{\forall \tau^i \in T}$
33: $prio_2 = modPrio(prio, \mathbb{S})$
34: $G^{c1}, G^{c2} = switchOrder(G^c, G^{backward}, G^{forward})$
35: $sQueue.push(\{G^{c1}, Obj^{new}, prio_1\})$
36: $sQueue.push(\{G^{c2}, Obj^{new}, prio_2\})$
37: **end while**
38: **return** $\mathbb{S}^{best}$

(i.e. the scheduling order of the tasks) (line: 11). We also adapted optimization 4 so that, when there are no conflicting tasks, the algorithm relies on the amount of contentions to modify the priority vector. However, relying on contentions in a more systematic way did not yield any improvement of the results.

At each iteration, the algorithm calls function $findSchedule$ (described in Algorithm 6 that we

detail later) to build a schedule $\mathbb{S}$ from scratch using a task system G^c, a vector *prio* that gives priorities to the tasks, and an objective Obj for the makespan of the schedule (line: 16). Once $\mathbb{S}$ is built, the algorithm compares its makespan with the makespan of the best schedule found so far: $\mathbb{S}^{best}$. If it is inferior, schedule $\mathbb{S}$ is saved as the new $\mathbb{S}^{best}$, the UB and LB are updated (line: 20) in order to lower the makespan objective in the next iteration, and changes are made to the task priorities to reflect the order of the starting dates of tasks in $\mathbb{S}$ (lines: 19-23). If the new schedule is longer than the best so far however, Obj is increased in order to give some more slack to the algorithm in the next iteration, and LB is increased as well if the algorithm has failed enough times (lines: 25-30). The algorithm then iterates, until either LB reaches UB or it runs out of new priority vectors to test.

There are several constants impacting the computation cost of the algorithm that are defined in an empirical way:

- Line 22: Obj^{new}, the next objective is set to $UB - 100$. The value must not be too ambitious to allow $findSchedule$ to find suitable schedules and the convergence towards the best priority vectors. As the minimum contention duration that we applied in our tests is 50 cycles, the number of contentions to avoid in order to improve the makespan is reasonable and 100 is also an order of magnitude below the duration of the tasks we scheduled who had a WCET superior to 1000 cycles (and sometimes superior to 20000 cycles).

- Line 26: $log2(|T|)$ bounds the number of consecutive attempts of $findSchedule$ without finding a better schedule than $\mathbb{S}^{best}$ before increasing LB. This bound must be high enough to let $findSchedule$ reach Obj but is also responsible for stopping the search when it is not possible. The number of tasks in the system impacts the size of the solution space. Our experiments are composed of systems ranging from 4 to 329 tasks so the $log2$ allows enough attempts for small systems while limiting them for the largest systems.

- Line 30: whenever a failure occurs, the objective is increased by at least 10% of its value (bounded by the current UB). This value has been kept from the original algorithm in [45].

One important point here is that the heuristic does not test all possible combinations of task priorities: at each iteration the current priority vector is modified, and the resulting vector is used in the next iteration if it has not already been used in a prior iteration. The way the algorithm modifies the priority vector does not guarantee that all priority combinations will be explored. In fact the objective of the heuristic is precisely to converge to a solution without having to explore all the combinations.

Algorithm 6 describes the $findSchedule$ function. This function iteratively creates a schedule $\mathbb{S}$ of the tasks of G^c on $\mathbb{C}$, using tasks priorities *prios* and an objective value Obj for the makespan of $\mathbb{S}$. A set of tasks ready to be scheduled (i.e. whose predecessors have already been scheduled) is maintained, and at each iteration, the ready task with the highest priority is selected for scheduling (line: 4). The selected task τ^i is scheduled following a given policy (in our experiments we used ASAP) and an interference analysis is performed on the resulting partial schedule $\mathbb{S}'$ (line: 6). Note that the priority vector does not define the mapping of the tasks so the scheduling policy is responsible for choosing the cores where tasks are scheduled. If $makespan(\mathbb{S}')$ is smaller than objective Obj, the algorithm updates the set of ready tasks and iterates with the next ready task (line: 18). If, however, the partial schedule spans more than Obj cycles, the algorithm is allowed to de-schedule some tasks that are put back in the set of ready tasks in order to make room for τ^i before Obj (line: 9). The de-scheduled tasks are the tasks that start after the end of the last predecessor of τ^i and before $Obj - WCET(\tau^i)$, as well as all their (already scheduled) successors. Tasks that start after this date and are not successors of de-scheduled tasks are not put back in the ready set, but are directly rescheduled following the ASAP

Algorithm 6 *findSchedule*

Require: G^c, $\mathbb{C}$, Obj, $prios$
1: $readyTasks = initRT(G^c)$
2: $budget = \alpha \times |T|$ ▷ *α is tuned according to the size of the task system*
3: **while** $(readyTasks \neq \emptyset) \wedge (budget > 0)$ **do**
4: $\tau^i = getNext(readyTasks, prios)$
5: $d = max_{\tau^h \in preds(\tau^i)}(\phi^h_{\Phi^h}.d^{\#})$
6: $\mathbb{S}' = scheduleASAP(\mathbb{S}, \mathbb{C}, \tau^i, d)$
7: $computeContentions(\mathbb{S}')$
8: **if** $makespan(\mathbb{S}') > Obj$ **then**
9: $resched, desched, \mathbb{S}_{temp} = unsched(\mathbb{S}, d, Obj, \tau^i)$
10: $readyTasks = readyTasks \cup desched$
11: **for** τ^j **in** $resched$ **do**
12: $\mathbb{S}_{temp} = scheduleASAP(\mathbb{S}_{temp}, \mathbb{C}, \tau^j, d)$
13: $budget = budget - 1$ ▷ *τ^j is scheduled again*
14: **end for**
15: $\mathbb{S}' = scheduleASAP(\mathbb{S}_{temp}, \mathbb{C}, \tau^i, d)$
16: $computeContentions(\mathbb{S}')$
17: **end if**
18: $updateRT(readyTasks, \tau^i)$
19: $budget = budget - 1$ ▷ *accounting for τ^i*
20: **end while**
21: **while** $readyTasks \neq \emptyset$ **do**
22: $\tau^i = getNext(readyTasks, prios)$
23: $d = max_{\tau^h \in preds(\tau^i)}(\phi^h_{\Phi^h}.d^{\#})$
24: $\mathbb{S}' = scheduleASAP(\mathbb{S}', \mathbb{C}, \tau^i, d)$
25: $updateRT(readyTasks, \tau^i)$
26: **end while**
27: $computeContentions(\mathbb{S}')$
28: **return** $\mathbb{S}'$

policy, in respect of their potential dependencies, in order to benefit from the free intervals in the schedule left empty by the de-scheduled tasks (lines: 11-14). Task τ^i is then scheduled (again, using ASAP in our experiments) (line: 15). Even if objective Obj is still unmet, the algorithm then goes on to the next task to schedule, hoping that further de-schedulings in the next iterations will allow to meet the objective. The de-scheduling of tasks significantly affects the execution time of the algorithm compared to a greedy solution, and can create an infinite loop under certain circumstances. In order to prevent it, an exploration budget (defined in line: 2) guarantees that the main scheduling loop does not iterate more than a fixed number of times, even though some tasks remain to be scheduled. If the number of iterations reaches the budget, the algorithm exits the loop and falls back to a greedy strategy (line: 21) for the tasks that remain to be scheduled. Tuning the budget value thus allows to trade execution time for precision.

We define a constant α (line 2) that sets the number of rescheduling operations allowed to reach the objective. We set $\alpha = 3$ for task systems with less than 26 tasks so that up to 78 tasks can be rescheduled and $\alpha = 1.2$ for the others which allows 394 rescheduling operations for the largest task system.

3.5.3.3 Merging Optimization

In certain situations, the multi-phase model may incur an overestimation of the number of contentions during the interference analysis. In the example depicted in Figure 3.13 (left), the yellow phase may contend with the three phases in parallel. As a result, the interference analysis counts 3 contentions coming from the yellow phase for each of these phases, resulting in 9 contentions in total. In practice this is impossible, as the yellow phase only performs 3 accesses in total. In order to reduce this pessimism, we developed a phase merging algorithm that can be applied on a partial or complete schedule. This optimization detects local situations in which merging together multiple phases of a task can reduce the overestimation of the number of contentions during the interference analysis.

In practice, the optimization looks for phases ϕ_j^i (called saturated phases in the following) that create more than $(|\mathbb{C}| - 1) \times \phi_j^i.m$ contentions to phases in parallel during the interference analysis. This formula was chosen as another trade-off between speed and precision. Once a saturated phase is discovered, the algorithm looks for phases scheduled in parallel and assesses whether or not it would be beneficial to merge them together. Indeed, the local benefits of merging phases (w.r.t. a given saturated phase) can be outweighed by the effects of the merge on adjacent tasks. This can be illustrated using Figure 3.13 :

- In the left part of the figure, the maximum number of contentions each phase may suffer is :
  - $min(5, x + 3)$ for the green phase.
  - $min(6, 3) = 3$ for the purple phase.
  - $min(4, 3) = 3$ for the red phase.
  - $min(x, 5)$ for the grey phase.
  - $min(3, (5 + 6 + 4)) = 3$ for the yellow phase.

  So if the phases are not merged, the interference analysis counts $9 + min(x, 5) + min(5, x + 3)$ contentions in total for the two cores.

- In the right part, when the phases are merged, this number is :
  - $min(15, x + 3)$ for the blue phase.
  - $min(x, 15)$ for the grey phase.
  - $min(3, 15) = 3$ for the yellow phase.

  So the interference analysis counts $min(15, x + 3) + min(x, 15) + 3$ contentions in total.

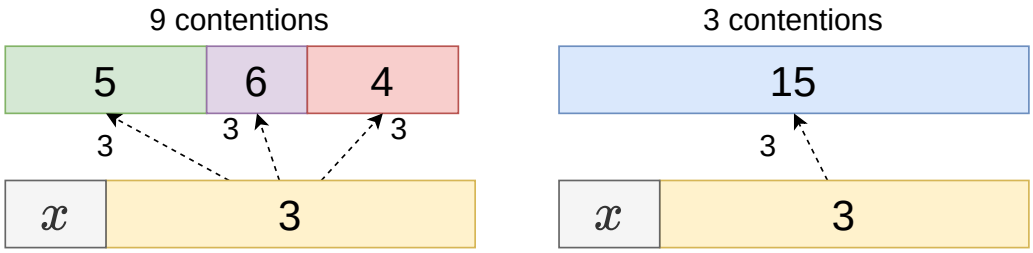


Figure 3.13: An example of local overestimation of contentions.

Therefore, if the value of x is strictly greater than 6, the merge is not globally beneficial.

Algorithm 7 describes the merging optimization. As for the SDE algorithm, computing the contentions several times is necessary to identify the saturated phases and to assess whether or not a merge is profitable. The algorithm retrieves the list of all scheduled phases and iterates over it until a saturated phase ϕ_j^i is found. When a phase is saturated, the algorithm enters the inner while loop (line 6) to try some merges. The merges are attempted using *candidates*, the list of phases in parallel of ϕ_j^i, that is retrieved by function *getPhasesWithin* (line 8). Then, function *getMergeablePhases* searches for two phases of *candidates* that are in the same task, consecutive and have not been studied before (otherwise they are present in *alreadyAttempted*). If no such phases have been found, the inner while is exited with a break (line 11). Otherwise, the phases are added to the *alreadyAttempted* list and a new schedule $\mathbb{S}'$ is created with the two phases ϕ_l^k and ϕ_{l+1}^k merged using function *mergePhases* that also recomputes the contentions. If the makespan of $\mathbb{S}'$ is better than $\mathbb{S}$ then the merge is confirmed at line 16.

The ASAP-based greedy heuristic described in Section 3.5.3.1 does not compute the contentions in the system before the schedule is produced. As a result, the scheduling decisions are not impacted by potential merges, so it is only useful to apply the merging optimization once the schedule has been entirely constructed. On the other hand, the SDE algorithm is interference-aware, so calling the merging optimization at each scheduling step can influence its decisions. In the remainder of the document, whenever the merging optimization is used, it is used after the schedule is produced with the ASAP policy, and during its construction with the SDE policy. We do not display the results of the optimization with IPH because it does not improve the trade-off between the computation speed and its efficiency to reduce the makespan of the schedule.

Algorithm 7 *mergeOptimization*

Require: τ^i ; $\mathbb{S}$; start; end
1: $phases = getPhasesIn(\mathbb{S}, start, end)$
2: $idx = 0$
3: **while** $idx < size(phases)$ **do**
4: $\quad\phi_j^i = phases[idx]$
5: $\quad alreadyAttempted = []$
6: $\quad$**while** $isSaturated(\mathbb{S}, \phi_j^i)$ **do**
7: $\quad\quad end = \phi_j^i.d^\# + \phi_j^i.dur + \phi_j^i.p$
8: $\quad\quad candidates = getPhasesWithin(\mathbb{S}, \phi_j^i.d^\#, end)$
9: $\quad\quad \phi_l^k, \phi_{l+1}^k = getMergeablePhases(candidates, alreadyAttempted)$
10: $\quad\quad$**if** $\phi_l^k == null$ **then** ▷ *no phases left that can be merged together in candidates*
11: $\quad\quad\quad$**break**
12: $\quad\quad$**end if**
13: $\quad\quad alreadyAttempted.push((\phi_l^k, \phi_{l+1}^k))$
14: $\quad\quad \mathbb{S}' = mergePhases(\mathbb{S}, \phi_l^k, \phi_{l+1}^k)$
15: $\quad\quad$**if** $makespan(\mathbb{S}') < makespan(\mathbb{S})$ **then**
16: $\quad\quad\quad \mathbb{S} = \mathbb{S}'$
17: $\quad\quad$**end if**
18: $\quad$**end while**
19: $\quad idx = idx + 1$
20: **end while**

3.6 Experimental Evaluation

In this section, we present a comparative study of the heuristics and an evaluation of the multi-phase model. Our evaluations use both synthetic tasks and task systems from real case studies.

3.6.1 Tests Metrics

The two metrics that we considered in the experiments are the *makespan* of the schedule in the presence of interference and the *total number of contentions* that may appear in the schedule according to the interference analysis.

For each metric m, the notion of *gain* is defined as the comparison of the value of m in a given schedule to the value of m in a baseline schedule (the single-phase variant of the schedule):

$$gain = (m\_value\_baseline - m\_value\_schedule)/m\_value\_baseline.$$

Moreover, a test is considered to be positive if $gain \geq 0$ for the corresponding task system, scheduling policy and metric m. In other terms, a positive test means that the multi-phase instance of the problem yields improved results compared to its single-phase counterpart.

3.6.2 Comparative Study Using the ILP Formulation

The optimization problem is inherently nonlinear due to the interference computation. Consequently, the ILP solver (Gurobi 9.5.1 [37]) encountered scalability issues. Thus, this section only presents the results for experiments with 4 to 6 tasks composed of 4 to 6 phases each, and whose solving time was inferior to 6 hours (a timeout was set in the experiments). The experiments include tasks with an average access rate of 25, 50 or 75 accesses per 10,000 cycles, without access over-approximation (the number of accesses in the single phase variant of a task is equal to the sum of accesses of the phases in the multi-phase variant), and with either no empty phases, or with 20% of the phases empty. In our experiments, we assume that the duration of accesses in isolation is 50 cycles. However, the memory latency of an access in the presence of interference can be several times the cost of an access in isolation due to indirect effects, as we saw in Chapter 2. The experiments were thus conducted with a penalty duration of 50 or 150 cycles, which correspond respectively to an optimistic and a more realistic architectural assumptions. Setting the penalty duration to 50 is indeed the most optimistic assumption for the target architecture, and usually is an unfavorable assumption for our experiments.

3.6.2.1 ILP results

The distribution of the gain values obtained by the multi-phase variants compared to their single-phase counterparts, scheduled with ILP is represented in Figure 3.14. The extreme values are not represented for readability reasons: the gain varies from -66.49% to 69.38% and the average value is 9.42%. In other terms, we can expect to reduce the system worst-case makespan by around 9% on average by switching from single to multi-phase representation. Moreover, in 96.19% of the tests the results of the multi-phase ILP were positive, i.e. at least as good as the single-phase ILP. As these results have been obtained with only small instances of the problem, they do not allow to draw general conclusions. However, it is worth noting that the gap between the two models tends to increase with the number of cores since the experiments with 2 cores have an average gain of 8.88% while it is 10.85% for the tests with 4 cores. This is coherent with the fact that the effect of potential contentions increases with the number of cores.

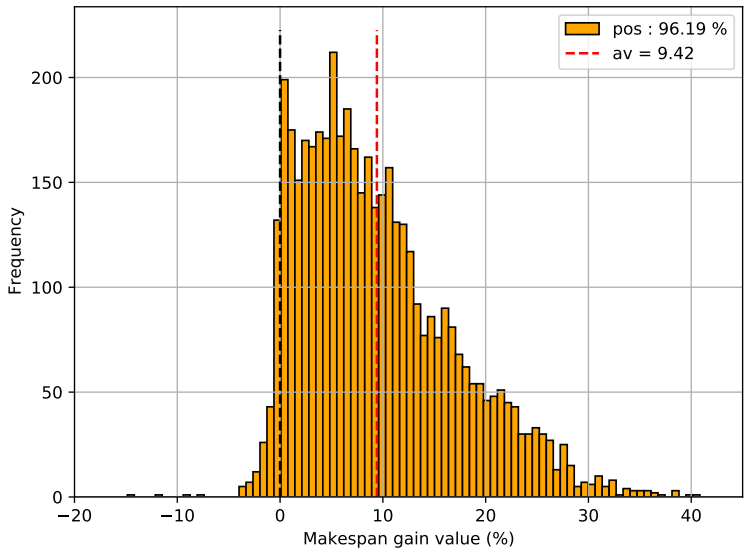


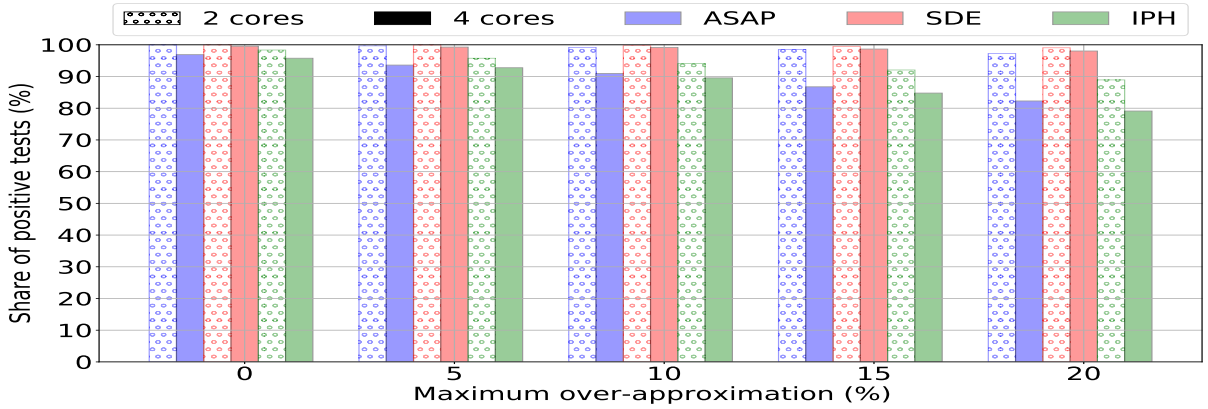
Figure 3.14: Makespan gain of multi-phase ILP vs single-phase ILP in %

Table 3.1: Makespan gain results compared to ILP with multi-phase or single-phase

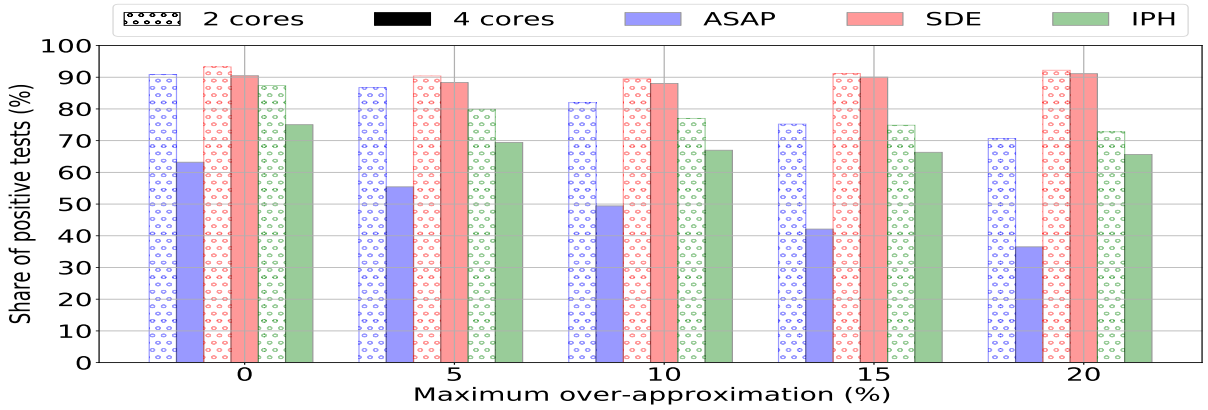
Cores	Heuristic	Gain vs ILP multi		Gain vs ILP single	
		Share of positive (%)	Average gain (%)	Share of positive (%)	average gain (%)
2	IPH	7.31	-3.71	90.51	5.17
	SDE	2.70	-5.69	73.77	3.20
	+merge	6.34	-5.05	77.47	3.83
	ASAP	2.15	-7.77	63.77	1.11
	+merge	5.15	-6.76	70.50	2.12
4	IPH	1.29	-5.02	88.06	5.82
	SDE	1.37	-5.77	81.29	5.07
	+merge	3.06	-5.43	83.87	5.42
	ASAP	0.56	-9.35	64.84	1.50
	+merge	1.69	-8.42	70.40	2.42

3.6.2.2 Comparison of ILP and heuristics for multi-phase profiles

Table 3.1 shows the average gain and the proportion of results where each heuristic result was at least as good as the ILP solving the multi-phase or single-phase problem. All the heuristics are on average less than 10% worse than the optimal multi-phase result and IPH is only at 3.71% on 2 cores and 5% on 4 cores. When the merging optimization was used, ASAP and SDE were even able to beat the multi-phase ILP (in respectively 2.5% and 3.3% of the tests) because of the new profiles generated by the optimization. A version of the ILP with possible merges would considerably increase its complexity and solving time so it is not proposed. Moreover, the heuristics applied on multi-phase profiles are at least as good as the optimal solution for the equivalent 1-phase profiles in at least 63% of the experiments (up to 90.51% for IPH with 2 cores). Finally, IPH finds the optimal multi-phase schedule in 7.31% and 1.29% of these (simple) experiments respectively on 2 and 4 cores.



(a) penalty = 50 cycles



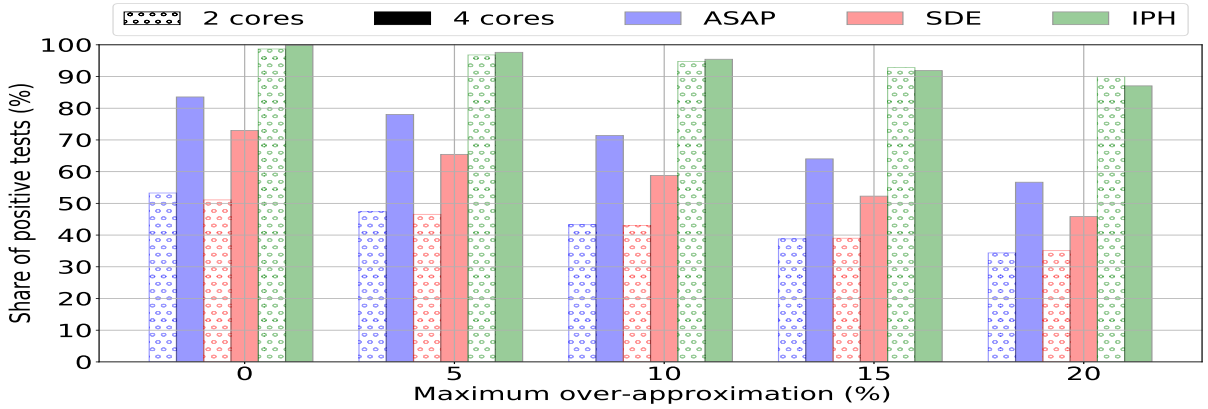
(b) penalty = 150 cycles

Figure 3.15: Share of positive results in terms of contentions according to the access over-approximation for 2 interference penalty values compared to single-phase IPH.

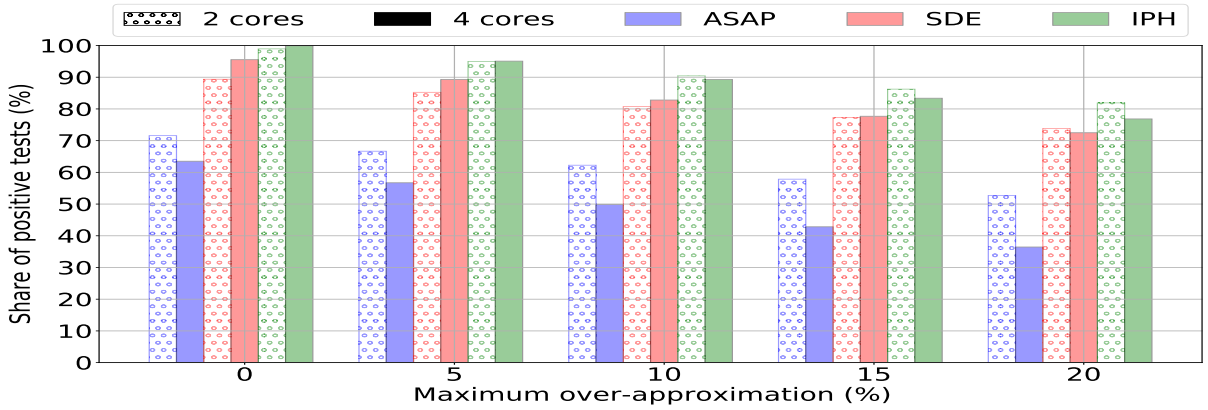
3.6.3 Comparative Study on Larger Systems

In this section we study the influence of the parameters on the gain of the multi-phase model and compare the heuristics on larger task systems. They are composed of either 20 or 25 tasks, with 15 or 20 phases on average. Moreover, several over-approximation values from 0 to 30% of additional accesses are tested. The other parameters are the same as in the previous section.

Figure 3.15 shows the share of experiments with a positive gain in terms of reduction of the worst-case number of contentions. It shows that SDE is the best heuristic to reduce the number of contentions, which is expected as it is the only one that makes decisions based on interference-aware (partial) schedules. In terms of makespan reduction, Figure 3.16 shows that IPH dominates the other heuristics. This confirms the results of [71]: tolerating a certain level of contentions in the system is more efficient, on average, than systematically postponing the start date of tasks to avoid interference, when it comes to reducing the makespan. However, when the penalty for a contention increases from 50 to 150 cycles, SDE improves, achieving results closer to IPH: as the penalty for each contention increases, postponing the execution of tasks to reduce the number of contentions becomes more profitable.



(a) penalty = 50 cycles



(b) penalty = 150 cycles

Figure 3.16: Share of positive results in terms of makespan according to the access over-approximation for 2 interference penalty values compared to single-phase IPH.

We started our experiments using ASAP as the baseline for single-phase as it was very fast. During the course of the experiments, we realized that IPH, although designed with multi-phase in mind, was also very efficient to schedule single-phase task systems, and outperformed ASAP in most cases. We thus decided to compare our multi-phase results with their single-phase counterparts scheduled with ASAP and IPH. The average makespan gain is represented by Figures 3.17 and 3.18 respectively against single-phase ASAP and IPH. The same observations as with the share of positive results can be made: SDE is the least efficient heuristic when the penalty is 50 cycles but its gain improves as the penalty increases. With a 150 cycles penalty, the gain of the multi-phase model reaches 15.86% using IPH against single-phase ASAP, while the maximum is 7.47% against single-phase IPH.

In a nutshell, IPH is the most adapted to reduce the makespan of the task systems while SDE is the most efficient to reduce contentions. The reason for this is that SDE tends to take short-term decisions that mainly reduce the worst-case number of contentions. However, it is sometimes better to accept more contentions locally to reduce the makespan of the entire system. When the effects of contentions are more important (i.e. a higher number of cores or a greater interference penalty), avoiding contentions is more correlated to reducing the makespan of the schedule so SDE becomes more efficient to reduce the makespan.

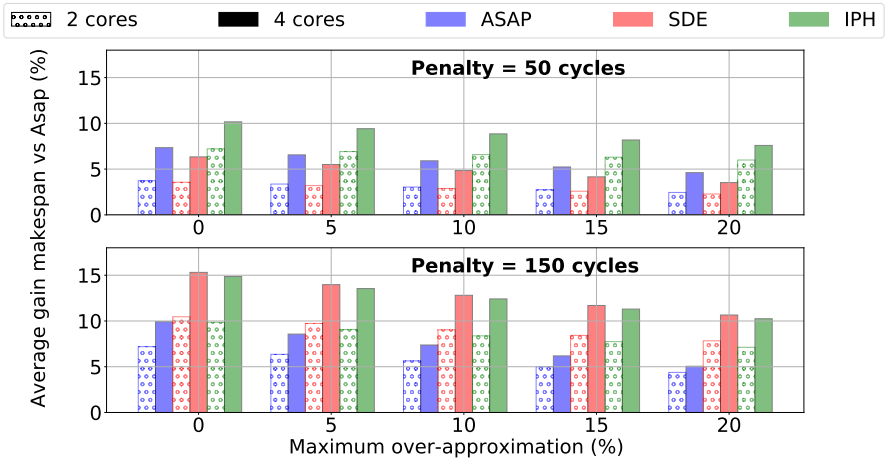


Figure 3.17: Average makespan gain vs ASAP single-phase according to the access over-approximation.

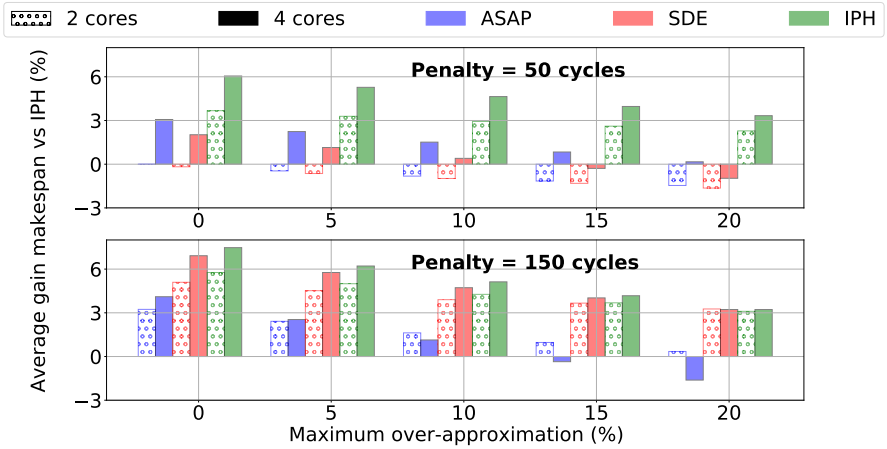


Figure 3.18: Average makespan gain vs IPH single-phase according to the access over-approximation.

Figure 3.19 shows the makespan gain of our three heuristics against IPH single-phase while varying the amount of empty phases (i.e. with no access). All heuristics perform significantly better when 20% of the tasks execution time is spent in empty phases, regardless of the value of the penalty, or of the level of over-approximation of accesses. When the penalty is set to 150 cycles, the difference in gain for SDE nearly doubles for 0% and 5% overestimation (and more than doubles for 15%). This demonstrates the crucial aspect of empty phases to improve the makespan of the scheduled systems.

Table 3.2 gives the average computation time of the heuristics according to the number of tasks in the system and the average number of phases. As expected, ASAP is the fastest of our 3 heuristics. Then SDE is faster than IPH for the systems composing our benchmark. However, when the workload increases the computation time increases comparatively more for SDE than for IPH. It is expected that SDE will be slower than IPH for very large systems.

3.6.4 Case Studies

In this section, we apply the heuristics on Rosace [62], a multi-periodic flight controller, and Papabench [60] that is derived from an open-source UAV control application. The environmental simu-

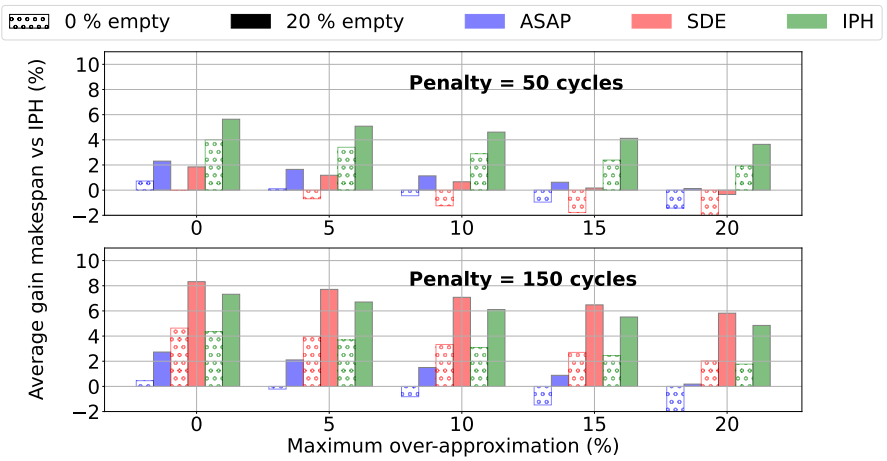


Figure 3.19: Average makespan gain vs IPH single-phase according to the presence of empty phases.

Table 3.2: Average computation time for the different heuristics

Tasks (#)	Phases per task (#)	Average computation time (s) ASAP	SDE	IPH
20	15	< 1	55	334
	20	< 1	112	499
25	15	< 1	88	596
	20	< 1	172	911
all tests		< 1	105	574

lation tasks of Rosace are not considered, as they are not embedded code and their WCET is several orders of magnitude larger than that of the other tasks.

The tasks have been compiled for ARM targets. We consider a multicore architecture in which each core has a private L1 LRU data cache and an instruction scratchpad. The memory latency was set to 50 cycles for non-cached accesses. The tasks were analyzed with OTAWA [9] to extract their CFG and perform a cache analysis. Then, the multi-phase profiles of the tasks were computed using the Time Interest Points (TIPs) method described in Section 3.4.

Some of the original tasks of Papabench were split to reduce the complexity of the analysis. Then, as the systems are multi-periodic applications, the task system was converted into a DAG of single-period tasks over one hyperperiod following the methodology of [19] but without using release dates for the jobs (i.e. with a synchronous release). The resulting DAG is composed of 77 tasks for Rosace and 329 tasks for Papabench. Statistics about the profiles are provided by Table 3.3 (*empty dur* is the proportion of execution time guaranteed without access). As expected, as δ diminishes, the

Table 3.3: Statistics of Rosace and Papabench profiles.

δ	Rosace sync (#)	Rosace phases (#)	Rosace ov-app (%)	Rosace empty dur (%)	Papabench sync (#)	Papabench phases (#)	Papabench ov-app (%)	Papabench empty dur (%)
1000	384	294	0.00	0.00	5 622	2 755	4.01	9.54
500	561	531	0.00	17.96	12 937	4 788	4.73	22.87
200	1 020	1 110	1.22	28.55	24 891	9 009	7.45	35.65

Table 3.4: Results of heuristics to schedule Papabench tasks with TIPs profile.

			penalty = 50 cycles			**penalty = 150 cycles**		
	nb cores	δ	gain makespan (%)		gain contentions (%)	gain makespan (%)		gain cont. (%)
			vs ASAP	vs IPH	vs ASAP	vs ASAP	vs IPH	vs ASAP
ASAP	2	1000	7.17	**-0.64**	8.32	9.15	3.37	17.58
		500	8.31	0.58	18.37	10.90	5.23	25.72
		200	**9.07**	1.41	24.71	**11.93**	**6.33**	28.96
	3	1000	4.81	1.18	14.18	6.13	1.60	7.32
		500	6.37	2.88	22.35	7.94	3.49	17.75
		200	6.96	3.41	23.68	9.00	4.61	13.18
	4	1000	**4.66**	1.25	19.85	**5.13**	**1.41**	13.96
		500	6.24	2.88	25.62	7.97	4.36	18.16
		200	6.84	**3.51**	27.43	9.01	5.44	19.61
SDE + merge	2	1000	-2.36	**-10.98**	47.13	9.94	4.21	57.06
		500	**0.96**	-7.38	54.73	**13.61**	**8.11**	64.70
		200	0.75	-7.61	40.59	12.85	7.31	62.83
	3	1000	**-5.05**	-9.06	30.50	7.44	**2.98**	59.58
		500	-1.39	-5.26	37.88	12.03	7.79	64.45
		200	-1.02	-4.86	38.46	9.88	5.53	68.35
	4	1000	-4.72	-8.48	36.38	**7.18**	3.53	65.11
		500	-1.61	-5.25	40.04	11.15	7.66	65.61
		200	-0.57	**-4.17**	38.90	9.83	6.28	67.17
IPH	2	1000	12.86	5.52	14.25	13.02	7.49	25.28
		500	14.18	6.95	25.85	**16.31**	10.99	35.08
		200	**14.97**	**7.81**	31.29	16.12	10.79	38.56
	3	1000	8.88	5.56	30.97	**10.96**	**6.67**	38.59
		500	9.98	7.36	39.56	14.91	10.80	37.37
		200	9.59	6.18	35.28	14.51	10.38	49.37
	4	1000	**7.80**	**4.28**	42.53	11.62	8.15	42.51
		500	9.42	6.26	44.13	13.83	10.44	47.20
		200	9.63	6.65	45.61	14.52	**11.16**	47.74

number of phases increases, reaching up to 14.4 phase per task on average for Rosace (resp. 27.38 for Papabench). The number of synchronizations required to implement these profiles remains under 2 per phase on average for Rosace, and reaches 3 per phase in the worst case for Papabench. The over-approximation also increases but remains low (less than 2% for Rosace and 8% for Papabench), while the percentage of time spent in empty phases increases fast, and reaches up to 28% for Rosace (resp. 35% for Papabench).

Then IPH, SDE and ASAP were used to schedule the DAGs on 2, 3 or 4 cores and we applied interference analyses on the schedules with a penalty of 50 and 150 cycles.

The results presented in Tables 3.4 and 3.5 show that the gain tends to increase when the phases are smaller (i.e. when δ is lower). Following our previous observations, this can be the result of the increased proportion of time spent in empty phases, and of a better distribution of accesses among phases. The multi-phase model globally yields better results than the 1-phase model, with a makespan gain up to 16.31% for Papabench (IPH on 2 cores with $\delta = 500$ cycles and a penalty of 150 cycles) and 24.00% for Rosace (SDE on 4 cores with $\delta = 200$ cycles and a penalty of 150 cycles). For Papabench, IPH always performs the best improvements, ranging from 7% to 16% compared to the 1-phase ASAP and between 4% and 11% compared to the 1-phase IPH. When the penalty is 50 cycles, SDE is the

Table 3.5: Results of heuristics to schedule Rosace tasks with TIPs profile.

			penalty = 50 cycles			**penalty = 150 cycles**		
	nb cores	δ	gain makespan (%)		gain contentions (%)	gain makespan (%)		gain cont. (%)
			vs ASAP	vs IPH	vs ASAP	vs ASAP	vs IPH	vs ASAP
ASAP	2	1000	**2.42**	0.76	9.03	**2.31**	1.28	3.80
		500	3.26	1.10	13.82	5.86	4.87	11.83
		200	4.90	2.77	22.35	9.15	**8.19**	18.93
	3	1000	4.71	**-0.04**	2.42	5.01	2.83	3.98
		500	6.96	2.32	9.79	6.48	4.33	7.12
		200	8.71	4.16	14.28	8.65	6.56	9.88
	4	1000	11.18	3.17	5.94	13.23	**0.83**	9.70
		500	13.65	5.86	11.59	14.89	2.73	13.22
		200	**15.78**	**8.18**	17.15	**16.80**	4.92	16.76
SDE + merge	2	1000	**0.90**	-1.32	15.82	10.75	9.81	39.66
		500	3.78	1.63	31.20	13.28	12.36	55.20
		200	2.85	0.67	37.11	17.04	**16.17**	50.69
	3	1000	1.11	**-3.82**	10.79	**9.39**	**7.31**	51.99
		500	5.17	0.44	23.82	12.34	10.32	44.12
		200	7.64	3.03	26.23	17.55	15.65	44.12
	4	1000	7.37	-0.99	14.58	20.36	8.98	55.80
		500	13.87	6.09	20.88	20.54	9.19	35.48
		200	**15.54**	**7.92**	26.31	**24.00**	13.14	40.11
IPH	2	1000	**4.26**	2.12	7.10	4.87	3.87	10.21
		500	5.70	3.59	14.32	9.20	8.24	21.86
		200	7.22	5.14	21.48	11.79	**10.86**	25.28
	3	1000	6.64	**1.98**	3.71	**5.19**	**3.01**	34.67
		500	9.05	4.52	10.54	8.95	6.86	24.95
		200	10.64	6.18	17.09	11.32	9.28	27.49
	4	1000	14.68	6.98	0.27	15.56	3.50	23.46
		500	17.35	9.90	7.27	17.82	6.08	45.18
		200	**19.04**	**11.73**	9.90	**19.69**	8.21	48.58

worst heuristic and its makespan is often higher than if the tasks are represented with the single-phase model (i.e. $gain < 0$). However, with a 150 cycles penalty per contention, SDE is more efficient than ASAP with a gain ranging from nearly 7% to 13% for the makespan. For Rosace, SDE is more efficient, as the gain is always positive and often close to ASAP with 50 cycles of penalty, and it is even the best heuristic when the penalty is 150 cycles.

The two tables also display the gain in terms of contentions. For Papabench (resp. Rosace), this gain ranges from 6.92% to 64.36% (resp. 2.42% to 55.80%) compared to 1-phase ASAP scheduling. This means that on top of reducing the makespan of the computed schedules, our heuristics, coupled with the multi-phase model, are able to significantly improve the timing predictability of the scheduled applications because there is less variability in the number of contentions that may occur in the system (i.e. the maximum interference scenario is closer to the average case scenario). SDE is the best heuristic to reduce contentions, even when it obtains negative makespan gains, which is coherent with what we observed with the synthetic systems.

With $\delta = 1000$ on 2 cores, the time required to schedule Papabench (resp. Rosace) with ASAP was 1 minute (resp. less than 1 second) while it took nearly 8 hours when applying SDE (resp. 43 seconds) and 6 hours (resp. 3 minutes) to run IPH with up to 31 threads (resp. 19) computing a

schedule at a time. However, as IPH is an iterative heuristic, it is able to find the best result or at least a satisfying result within the early iterations. For Papabench the schedule was found in less than 3 hours.

3.7 Conclusion

This chapter presented the multi-phase task model as a solution to tighten the worst-case estimation of interference effects in multi-core architectures. We started by introducing a formal model of the multi-phase representation of tasks, and presented a set of properties that guarantee the correctness of such a representation w.r.t. the corresponding task execution. In particular we showed that in order to maintain coherence between the multi-phase model and the actual execution of the task, once the interference analysis has been performed, synchronizations must be included in the tasks code and the synchronization dates must correspond to the schedule dates computed during the interference analysis. Otherwise, it may happen that some memory accesses are performed in time windows corresponding to phases for which they were not accounted for, thus invalidating the results of the interference analysis.

In order to obtain multi-phase representations of tasks, we then described the Time Interest Points framework, which consists in a series of models and analyses that take as input the binary code of the tasks and outputs their multi-phase representation. Since the performed analyses are complex, the framework abstracts the CFG of the tasks in a simplified structure called a TIPsGraph that only expresses the control flow between instructions that may perform memory accesses, and thus generate or suffer from interference. From the TIPsGraph of a task, our algorithms then enumerate its abstract execution traces and generate intermediate phases according to the behavior of the enumerated traces. Finally, the intermediate phases are mixed together to obtain the profile of the task.

The last part of the chapter was dedicated to the presentation of scheduling algorithms for the multi-phase model, and to their evaluation. We started by presenting an ILP formulation of the problem. Since the ILP was too complex to solve reasonably large problems in acceptable time, we also presented three heuristics. The fastest one is based on ASAP scheduling and does not base its decisions on the amount of generated interference. A more subtle heuristic (SDE) verifies if postponing the start date of a task may improve the schedule by reducing the amount of interference. In practice, it is very efficient to reduce the overall amount of worst-case interference, but this reduction only is beneficial to the makespan of the system if the time penalty for each interference is large enough to compensate the postponing of the start date of the tasks. Finally, the third heuristic we presented was inspired by the work of [45] and [55]. This algorithm iteratively reduces the makespan of the system by automatically switching the order in which the tasks are considered for scheduling, and by applying a series of local optimizations. This heuristic is very efficient to reduce the makespan of the scheduled task systems. Moreover, it is able to generate efficient schedules after only a few iterations, which makes it suited for larger systems. Our evaluation also shows that on average, on small systems scheduled with the ILP, the expected gain of switching from single to multi-phase was around 9%. On our two more realistic applications, switching to multi-phase yields gains of over 10%. This study validates the approach and the model, and paves the way for future improvements.

Regarding the contents of this chapter:

- these results led to the submission and acceptance of the ANR JCJC MeSCAliNe project (2022-2026) that regards the predictable implementation of neural-networks in real-time systems for autonomous vehicles, with me as project leader.
- these results allowed us to contribute to the submission and acceptance of the CAOTIC ANR

project (2023-2026) led by Claire Maïza and Lionel Rieg, that regards interference in parallel architectures (multi-cores, many-cores and GPUs). As part of this project, I am currently collaborating on other aspects of the multi-phase model with Isabelle Puaut (IRISA, Rennes) and Hugues Cassé. This collaboration is centered around the PhD thesis of Hector Chabot (started October 2023), directed by Isabelle and co-advised by Hugues and myself.

- the TIPs framework was developed collaboratively with Hugues Cassé. A first version of the framework was presented in the WCET workshop in 2018 [13]. The current version of the framework was presented in the ARCS conference in 2021 [14]. The work presented in this manuscript regarding the TIPs framework was adapted from this last publication.

- the work regarding the multi-phase model, including the correctness criteria, the scheduling algorithms and the evaluation, were part of the PhD thesis of Rémi Meunier. This thesis was directed by Thierry Monteil and myself. As a CIFRE thesis it was partially funded by Randstad digital and ANRT, and was also partially funded by the JCJC ANR MeSCAliNe project. Rémi successfully defended his thesis on November 30^{th}, 2023.

- the results regarding the correctness criteria were published in ECRTS 2022 [57].

- the results regarding the scheduling algorithms and the evaluation were submitted to the Real-Time Systems journal, and are currently under review.

- as part of the CAOTIC project, I am also co-directing the PhD thesis of Louison Jeanmougin (started October 2023), together with Christine Rochange and Houssam-Eddine Zahaf (LS2N, Nantes). This thesis targets interference in GPUs, and one of its aspects is to investigate how the multi-phase model can be extended to handle the specificities of GPU kernels. Preliminary results related to this thesis were obtained during the Master 2 internship of Louison (2023) and have been published in the WCET workshop in 2023 [48].

Chapter 4

Conclusion and research project

4.1 Conclusion

This manuscript presents the main research results I have been involved in regarding parallelism issues in the timing predictability of real-time systems, since I arrived in Toulouse in 2016. These results focus on two aspects:

- dealing with parallelism issues at the instruction level to avoid timing anomalies and guarantee the timing-compositional behavior of CPU cores while maintaining a high level of performance. This aspect led to the design of the MINOTAuR core.
- dealing with parallelism issues at the task system level to tighten the estimation of interference effects, while assuming the CPU is composed of timing-compositional cores. This aspect was addressed by introducing the multi-phase model, providing correctness criteria for its implementation, developing the TIPs framework to construct multi-phase representation of tasks, and designing static scheduling algorithms.

This work was for me the occasion to conduct original research and to co-direct my first PhD students following my own directions. These directions are diversified and complementary, ranging from architecture design to multi-core static scheduling and static analysis of binary code, and were tackled using formal models and methods at the core of the solutions. In the end, the presented results constitute a holistic effort to make the static analysis of real-time systems possible, safe and precise for multi-core targets.

4.1.1 Other research directions

Since I arrived in Toulouse, I was involved in other research directions that I did not present in the manuscript. I will now briefly describe them.

Static timing analysis for GPUs: The rapid adoption of machine learning techniques for autonomous embedded systems such as unmanned air and ground vehicles pushes towards the use of GPUs in real-time embedded systems. This raises many issues, since all the problems described and tackled in this manuscript exist in GPUs, but GPUs also add problems of their own. The Single Instruction Multiple Threads (SIMT) execution paradigm introduces a phenomenon called thread divergence that has a strong impact on the timing behavior of the execution of GPU kernels. In [48], we showed that it was possible to deal with it by creating warp-level CFGs that model the effects of

thread divergence inside said warps, while remaining compatible with the traditional static WCET analysis methods. Since then, Louison Jeanmougin has been working (as part of his PhD thesis (2023 -), co-directed with Christine Rochange and Houssam-Eddine Zahaf) on a way to safely and precisely combine the execution profiles of the various warps executing a kernel on the same GPU. In this work, the profiles are described using the multi-phase model to represent alternating periods of execution on the GPU and of waiting for the result of long latency operations.

GPU-specific interference: As part of the JCJC ANR MeSCAliNe project, Noïc Crouzet has been working on the design of a predictable GPU architecture, since his Master 2 internship (2023) and the start of his ongoing PhD thesis (2023 -), co-directed with Christine Rochange. Starting from an open-source GPU design based on an extension of the RISC-V ISA, we encountered a serious threat to timing predictability. GPUs handle warps by implementing multiple hardware queues to store the decoded instructions of each warp separately. At each cycle, a warp scheduler is responsible to elect a warp that can issue and execute an instruction. For timing analysis purposes, it is desirable that the warp scheduler policy be predictable and known. In the literature, the warp schedulers in Nvidia GPUs have been described as implementing the Greedy then Oldest or Greedy then Round Robin predictable policies, depending on the generation of the GPU family. However, our preliminary work shows that the actual issuing policy of the GPU can be made unpredictable depending on how the instructions are fetched: indeed, in the fetch stage another scheduler chooses for which warp the next instruction is going to be fetched. Depending on these choices, the contents of the instruction queues may vary, and can reduce the options for the warp scheduler at the issue stage, rendering its actual choices unpredictable.

Efficient and certifiable neural network inference implementation: In the 2020-2024 period, I have been collaborating in an ANITI[1] chair entitled: Towards the certification of ML-based systems. I was involved in the co-advisement of the PhD thesis of Iryna de Albuquerque Silva (2021 -), directed by Claire Pagetti from ONERA and co-advised with Adrien Gauffriau from Airbus. In her thesis, Iryna has developed a compilation framework called ACETONE that takes as input a representation of a trained neural network and produces a C implementation of the inference function for this network, with properties that allow its certification (e.g. traceable code, no compiler optimizations, no dynamic memory allocation, etc.). The framework was first presented in ECRTS 2022 [22]. It was then extended to properly handle convolution layers, and published in the Real-Time Systems journal [23]. More recent work have tackled the optimized yet predictable implementation of convolution layers, based on high-performance computing (HPC) algorithms [21]. I have been contributing mainly to the timing predictability aspects of this work.

4.2 Research project

The results presented in this manuscript lay the bases for future research directions that I present in this section.

4.2.1 Extensions of the MINOTAuR core

The MINOTAuR core will be extended in several directions with multiple objectives in mind: (1) pursuing the construction of an efficient and predictable core by including more complex acceleration mechanisms, (2) extending the support for more instructions of the RISC-V ISA, and enabling key

[1] ANITI is one of the French 3IA Artificial Intelligence Interdisciplinary Institutes.

mechanisms for OS support, in order to use MINOTAuR as an experimental platform for other research (including research of the Real-Time systems community) and (3) building a multi-core framework in order to experiment with architectural design ideas (e.g. interconnects, arbitration cores) and with analysis solutions (e.g. multi-phase).

First, I will investigate how to enable the Memory Management Unit (MMU) so it does not generate timing anomalies. I do not foresee any major difference compared to what we have already done in the core, so this step should be quick and relatively easy. However, it is crucial in order to make the core Linux-ready, which I believe will be beneficial to the Real-Time research community. This work could be the topic of a Master 1 or 2 internship this year.

Second, the MINOTAuR core will be extended to a predictable superscalar design with out-of-order execution. The main challenge is that the definition of monotonicity that is at the heart of our proofs of absence of timing anomalies was designed with in-order execution in mind and is too restrictive to be used for out-of-order execution. In essence, out-of-order execution can create situations in which monotonicity is broken for a short period of time and in a limited portion of the pipeline. However, by reordering instructions in the commit stage, and with a clever design of the issue and execution stages, it is possible to guarantee that monotonicity is restored once the instructions leave the pipeline. We thus need to relax the definition of monotonicity to a version that complies with the specifics of out-of-order execution, and then to find general design guidelines that make a core provably timing predictable. This will be tackled as part of the remainder of the PhD thesis of Alban Gruin.

In parallel, a predictable multi-core platform based on MINOTAuR will be designed. The first aspect of this work will be to extend the supported list of instructions to include multi-core synchronization support. The challenge will be to explore efficient designs that do not generate interference or for which the contentions can be safely bounded. A second aspect will tackle the implementation of predictable memory access protocols enforced by the bus and memory controllers. A starting point will be to look at existing work regarding ARM MPAM protocol [79] and the quality of service configurations of the Xilinx UltraScale+ system-on-chip [29] and to try to adapt these protocols to directly target predictability while maintaining sufficient performance levels. The last aspect regards the design of a core dedicated to the orchestration of memory copies between the shared memory and the local private memories of the cores. This last item is the topic of a CIFRE PhD thesis in collaboration with Continental (we are currently looking for candidates).

Finally, the MINOTAuR core will be modelled in OTAWA. Building on the acquired experience and a good knowledge of the core, it should be reasonably simple to model its timing properties with a high precision in an OTAWA model, which is usually an issue for commercial processors. Doing so will provide a new experimental platform, in which it will be possible to conduct measurements on the synthesized version of the core and analyses with OTAWA on the same core. Moreover, building an OTAWA model for MINOTAuR, opens the possibility to derive multi-phase representations of tasks for this target, once the MINOTAuR-based multi-core architecture is built. This work could constitute the topic of a Master 1 or 2 internship in the close future.

4.2.2 Proof automation

The formal proofs of monotonicity enumerate many sub-cases, and it is easy to forget or get one wrong when performing the proof by hand. For the same reason, reading these proofs to convince oneself of the monotonicity of a core is also a difficult task. To avoid these shortcomings, we started using the Coq proof assistant to make sure that no sub-case was missing. However, writing a new Coq model and proof for each new core or each variant of a core is also time-consuming. On the other hand, in the first-order logic model, the pipeline is depicted as an acyclic graph whose nodes have a limited set of properties. For a family of processors (e.g. in-order pipelines), the shape of the graph and the

actual properties of the nodes may vary, but the set of properties remains the same. As the proofs rely on these shapes and properties in a mechanical way, I wish to explore how to automatically generate a Coq model from the first-order logic model of a core, and to investigate to what extent the proof itself can be automatically generated. In the meantime, the B method will also be employed to prove the monotonicity of a particular pipeline design as a refined property of the monotonicity of more abstract pipeline models. Both solutions (Coq and Event-B) will be explored as part of the ANR ProTiPP project, in collaboration with colleagues from the ACADIE team at IRIT (Mamoun Filali, Jean-Paul Bodeveix).

4.2.3 Extensions of the multi-phase model

The first aspect of this research direction concerns the improvement of the TIPs framework, by combining its methodology with the one of StAMP [24], developed in Rennes. Each method has its own advantages and drawbacks. The TIPs approach offers more precision and control, by working on the explicit execution traces and by placing synchronizations at the necessary locations in the code. However, enumerating the traces may be intractable for complex applications, and adding too many synchronizations in the code may be unfeasible for embedded targets with limited memory. On the other hand, the StAMP approach relies on the Implicit Path Enumeration Technique, and thus can handle complex software with many possible execution traces, and synchronizes the code only on very specific locations, that are statically guaranteed to be on any execution path. This greatly limits the number of required synchronizations, but also dramatically reduces the options in terms of multi-phase profile design. One aspect of the PhD thesis of Hector Chabot is to investigate how to combine the best of these two worlds in order to limit the drawbacks of each method while enjoying their benefits. One key aspect lies in the limited use of Best-Case Execution Time (BCET) to improve the precision of the profiles while keeping the number of required synchronizations low. On top of designing new static analysis methods to derive this information, this approach requires to redefine the correctness criteria (and the underlying method to correctly count the number of accesses per phase) that we described in this document, in order to make them fit the specifics of the new model. Indeed, the swap from synchronizations to BCET information in order to lower-bound the execution dates of the memory accesses imposes the design of a new interference analysis method, since without synchronizations, adding a penalty to a phase in the model has no concrete effect on the corresponding code. It could thus happen that the assumptions made in the interference analysis no longer correspond to the actual code of the task. The interference analysis must be extended to take this into account.

Another promising aspect would be to exploit the multi-phase model in online (limited) preemptive scheduling algorithms. Indeed, phase boundaries make great candidates for preemption points, and cache effects can be computed at the granularity of phases, by adapting classical CRPD methods [2]. All synchronizations would be handled by a (global or core-local) scheduler or dispatcher that would be responsible for orchestrating the release of phases. The main challenges lie in the definition of efficient online scheduling algorithms adapted to the multi-phase model and in the implementation of the scheduling/dispatching code whose accesses to the shared memory and buses must remain limited and statically bounded. Such a solution also opens opportunities to place synchronizations inside (nested) loops, which corresponds to a current limitation of the TIPs and StAMP methods. Lifting this limitation will allow the representation of loop iterations (or groups of iterations) as separate consecutive phases in software that is dominated by loops, such as embedded neural networks.

4.2.4 Static timing analysis for GPUs

Regarding GPUs, I wish to pursue and build on the preliminary research results we have obtained. This requires pursuing two complementary objectives: (1) building a static WCET analysis framework for GPU kernels, on the same model as for CPUs, and (2) designing predictable GPU pipelines that allow simplifying hypotheses in the analysis framework. The analysis framework must include a micro-architectural model of the GPU pipeline in order to derive a multi-phase representation of the worst-case execution of a single warp in isolation. Each phase must account for the potential number of accesses made to each memory of the GPU separately in order to compute precise interference on each memory subsystem (global/local and shared), and on whether it corresponds to a phase of active work with instructions being issued for the warp, or a phase of waiting for a long latency operation. Being able to derive such a representation is the first objective of this research direction. The second objective is to derive a conservative yet precise formula to combine the representations of warps belonging to the same thread block. A first step will be to assume that all warps in the same block have the same worst-case multi-phase profile. This formula can then be complemented by adding or subtracting interference terms that account for the cache effects of each warp on the execution of the others, as well as contentions in the memory subsystems. The PhD thesis of Louison Jeanmougin, as part of the CAOTIC ANR project, will be the occasion to start working on this direction, which will be pursued in the long term.

This analytical approach can be seen as the GPU equivalent of the compositional approach in CPUs, and in the same fashion, it requires the GPU pipeline to operate in a time-predictable fashion. This requires the design of efficient and predictable components for the GPU pipelines: the same problems that exist for CPUs (e.g. parallelism between instruction and data caches) are present in GPUs, in addition to original ones (e.g. a single fetch unit shared by all warps). These GPU-specific issues must be identified, and for each of them a solution must be found, may it be a modification of the design, of the assumptions made in the analysis, or a combination of both. This will be the topic of the PhD thesis of Noïc Crouzet, as part of the MeSCAliNe ANR project, and will also be pursued in the middle to long term.

4.2.5 Predictable, parallel implementation of neural network inference

The last research direction I intend to pursue is related to the parallel implementation of neural networks. Currently, the inference function of neural networks follows HPC algorithms that try to optimize the placement of the tensors in the various caches of the memory hierarchy, in order to maximize the reusability of loaded blocks. As it happens, these algorithms are designed to be highly predictable (at least in their single-core versions), and in fact it is this predictability that is being exploited in order to increase the performance of the inference function. However, the commonly used implementations [76, 74] rely on unpredictable mechanisms such as dynamic memory allocation that preclude their use in a time-critical context. The work that has been started with ACETONE must be pursued to demonstrate that it is feasible and desirable to reconcile performance and predictability for neural network inference in embedded systems. The next steps in this effort are related to (1) the predictable parallel implementation of neural network layers, in particular by adapting highly optimized general matrix multiplication (GEMM) algorithms, (2) the extension of the list of supported layer families and network architectures in ACETONE and (3) the extension of ACETONE to GPU targets. The first item is an extension of our latest work on predictable optimized GEMM functions for single-core targets. The main difficulties in switching to multi-core lie in the multiple interference sources: contentions on the buses/memory controllers, such as the ones studied in this manuscript, and cache block invalidation in shared L2 or L3 caches. These events must be bounded by a combination of

a precise knowledge of the architecture and algorithm parameters to tackle the blocks invalidation issue, and of the usage of a practical abstraction such as the multi-phase model to tackle the interference on the buses. This research will start in the second semester of 2024 with a M2 internship and will be pursued along with the two other items in a CIFRE PhD thesis (starting 2025) as part of the ANITI chair I collaborate in.

Detailed Curriculum Vitae

État Civil

Nom :	Thomas Carle
Né le :	29/05/1987
Nationalité :	Française
email :	thomas.carle@irit.fr
web :	https://www.irit.fr/~Thomas.Carle/

Postes occupés

2016 -	**Maître de conférences** à l'Université Toulouse 3 Paul Sabatier
2016 - 2016	**Postdoctorat** à l'IRT SystemX (Palaiseau)
2015 - 2016	**Postdoctorat** à Brown University (Providence, RI, USA)

Formation

2011 - 2014	**Doctorat** à INRIA Paris (Univ. Paris VI Pierre et Marie Curie) *Soutenu le 31/10/2014. Directeur : Dumitru Potop-Butucaru.* *Titre : Efficient compilation of embedded control specifications with complex functional and non-functional constraints.*
2009 - 2011	**Master of Science** à Chalmers University (Göteborg, Suède) *Secure and dependable computer systems*
2007 - 2011	**Ingénieur civil des Mines** à Mines Nancy *Spécialité informatique*

Enseignement

2016 -	Mes enseignements à l'université se font à la Faculté de Sciences et d'Ingénierie (FSI) dans le département informatique. J'enseigne principalement l'architecture des processeurs du niveau L1 au niveau M1, ainsi que d'autres sujets en lien avec le parallélisme (programmation multi-coeurs en OpenMP, cohérence des caches, CUDA) au niveau L3 et M1.

Enfin j'interviens en M2 sur des thématiques en lien avec le temps-réel (langages synchrones, analyse statique et géneration d'exécutables embarqués temps-réel). Au début de ma carrière, j'ai également enseigné l'algorithmique et la programmation C et Python en licence, ainsi que la compilation en M1.

Participation à des projets de recherche

2024 - **ANR ASTRID Printemps :**
Ce projet est réalisé en collaboration avec des ingénieurs de recherche de Thales TRT. L'objectif est de développer une plateforme multi-coeurs prédictible temporellement et incluant des mécanismes matériels de sécurité.

2023 - **ANR CAOTIC :**
Ce projet rassemble des collègues de Grenoble (Verimag), Nantes (LS2N), Toulouse (IRT Saint Exupery), Paris (INRIA, CEA, Télécom Paris). L'objectif est d'étudier, de quantifier et si possible de réduire l'interférence pire-cas dans les systèmes temps-réels embarqués sur des architectures multi-coeurs et GPUs.

2022 - **ANR JCJC MeSCAliNe (Responsable) :**
Ce projet concerne la mise en œuvre sûre et temporellement prédictible d'applications de réseaux de neurones pour les véhicules autonomes. Il s'agit de développer des modèles et des techniques d'analyse ainsi que des optimisations dédiés aux réseaux de neurones et aux architectures matérielles capables de les exécuter de manière efficace (CPU multi-coeurs et GPUs). Dans le cadre de ce projet, un doctorat et un post-doctorat de 2 ans vont être financés.

2022 - **ANR ProTIPP :**
Ce projet a démarré en automne 2022 et rassemble des enseignants-chercheurs de l'équipe TRACES et de l'équipe ACADIE de l'IRIT. L'objectif de ce projet est de développer des processeurs informatiques temporellement prédictibles et performants, et de prouver formellement leur prédictibilité.

2022 - **Labex CIMI COCOON :**
Dans ce projet, l'objectif est de réfléchir sur les différents modèles existants pour représenter un processeur (depuis le langage de spécification du processeur comme VHDL, jusqu'à des langages très spécifiques permettant d'exprimer certaines propriétés temporelles), et de fournir des outils et des méthodes permettant d'accroitre la confiance que l'on peut avoir dans le fait qu'un modèle donné représente effectivement le comportement d'un processeur donné.

2019 - 2021 **Labex CIMI AVATAr (Responsable) :**
Ce projet rassemblait 4 enseignants-chercheurs permanents de mon équipe pour une durée de 2 ans, afin d'investiguer le comportement temporel des GPUs Pascal de NVIDIA, dans la perspective de construire un modèle d'analyse de pire- temps d'exécution.

Encadrement de thèses

2020 - 2023

Rémi Meunier. Soutenue le 30/11/2023. Thèse CIFRE (Randstat digital)
Co-encadrée à 50% avec Thierry Monteil
Rapporteurs : Angeliki Kritikakou (IRISA) et Emmanuel Grolleau (ENSMA)
Titre : Prédiction du temps d'exécution d'applications dans des architectures multi-cœurs.
Résumé : Cette thèse explore la thématique du modèle multi-phases de tâches pour réduire la surestimation des interférences entre tâches s'exécutant en parallèle sur des architectures multi-coeurs. Trois axes principaux sont abordés : (1) la formalisation des critères de correction du modèle multi-phases, (2) l'obtention et l'amélioration des profils multi-phases, par le biais d'algorithmes génétiques et d'heuristiques ad-hoc, et (3) l'ordonnancement statique des profils multiphases.

2023 -

Louison Jeanmougin. Thèse financée par projet ANR (CAOTIC)
Co-encadrée à 33% avec Christine Rochange et Houssam-Eddine Zahaf (LS2N)
Titre : Analyse de Pire-Temps d'Exécution sur Architecture GPU
Résumé : Le calcul de WCET pour des applications exécutées sur des processeurs monocoeurs est une discipline mature. Cependant, les techniques existantes ne peuvent être utilisées directement dans le contexte des GPUs : afin de gérer des milliers de threads actifs en parallèle, les GPUs adoptent un modèle d'exécution très différent et sensiblement plus complexe que celui des CPUs. Le premier objectif de cette thèse est de fournir une solution analytique à une instance du problème dans laquelle un certain nombre d'hypothèses simplificatrices seront faites (un seul bloc, warps symétriques, architecture simplifiée, etc.) : l'objectif est d'obtenir une formule (ou un ensemble de formules) aussi simple que possible permettant de caractériser les effets temporels liés à la concurrence entre warps d'un bloc. Par la suite, cette formule sera complexifiée en relaxant les hypothèses simplificatrices, étape par étape, pour prendre en compte notamment les différentes sources d'interférence matérielle entre threads et entre warps concurrents. Ces sources d'interférences et leurs effets sur le temps d'exécution pourront être caractérisés dans un premier temps par le biais de mesures.

2023 -

Noïc Crouzet. Thèse financée par projet ANR (MeSCAliNe)
Co-encadrée à 50% avec Christine Rochange
Titre : Architecture GPU open source prédictible

Résumé : Les systèmes embarqués temps réels critiques sont omniprésents dans l'industrie du domaine du transport. Avec l'évolution de la technologie, de nouvelles utilisations ont vu le jour : des algorithmes de détection visuelle ou de traitement de signaux sont maintenant réalisables grâce à l'utilisation d'architectures parallèles. Cependant, dans l'objectif de garantir les contraintes des produits et pour satisfaire les autorités de certification, ces technologies ne sont réellement utilisables que si l'on est capable de comprendre suffisamment ces architectures pour en fournir une modélisation permettant de calculer une borne WCET pour ces applications. Une étude d'un design de GPU simple et open source représente un point de départ solide pour mettre au point des techniques d'analyse statiques applicables aux architectures parallèles. Le GPU Vortex est issu d'un projet de recherche académique reconnu, il est suffisamment simple pour servir de point de départ aux recherches tout en implémentant les fonctionnalités de base attendues d'un GPU. L'objectif initial de la thèse est d'arriver à un modèle complet permettant de calculer statiquement un pire-temps d'exécution pour un kernel donné exécuté sur cette cible. Ce calcul se basera sur une abstraction des mécanismes d'exécution du GPU Vortex. Par la suite, des modifications pourront être apportées au design du GPU afin de le rendre plus prédictible (réduire le fossé entre l'abstraction et l'exécution réelle) et plus performant (ajouter des mécanismes plus avancés, tout en conservant la prédictibilité).

2023 - Hector Chabot. Thèse financée par projet ANR (CAOTIC)
Co-encadrée à 33% avec Isabelle Puaut (IRISA) et Hugues Cassé
Titre : Fine grain software modeling and analysis for interference management in multi-core real-time systems
Résumé : Les interférences au sein des tâches temps-réel sont souvent représentées sous la forme d'une unique valeur comptant le nombre maximum d'accès à des ressources partagées. Cependant, les accès aux ressources partagées ne sont pas réparties uniformément au long de l'exécution des tâches. L'objectif de cette thèse est d'améliorerla capacité des analyses de logiciel à estimer et réduire les interférences dans les architectures multi-coeurs en cassant la représentation monolithique de l'exécution des phases et en la remplaçant par le modèle multi-phases. Un des défis sera d'identifier la granularité optimale des phases en fonction du comportement et du code de chaque tâche.

2021 - Iryna De Albuquerque Silva. Thèse CIFRE (ANITI – Airbus)
Co-encadrée à 33% avec Claire Pagetti (ONERA) et Adrien Gauffriau (Airbus)
Titre : Environnement de programmation certifié pour les applications de machine learning.

Résumé : L'apprentissage automatique gagne une considération importante dans le domaine des systèmes critiques, y compris en aéronautique. Cependant, comme ces applications n'atteignent pas les niveaux de confiance de sécurité classiques et ne sont pas mises en œuvre avec un processus de développement accepté, de nombreuses activités de recherche et d'ingénierie doivent être menées avant de les intégrer dans les avions. La question de savoir comment implémenter en toute sécurité et de manière fiable un réseau neuronal sur un matériel adéquat est d'une importance vitale. En effet, les exigences de certification, en particulier celles de la DO 178C, imposent de fortes garanties sur la qualité du code et attendent du concepteur qu'il calcule le WCET. Le but de la thèse est d'explorer ces problématiques.

2021 - Alban Gruin. Thèse sur bourse ministérielle
Co-encadrée à 50% avec Pascal Sainrat
Titre : Cœur de calcul RISC-V efficace, déterministe et composable.
Résumé : Cette thèse explore la conception d'architectures matérielles prédictibles efficaces et complexes, en partant du coeur RISC-V MINOTAuR et en introduisant petit à petit des mécanismes d'accélération de plus en plus aggressifs (store buffer, superscalarité, stations de réservation, etc.). En s'appuyant sur la notion de monotonicité du progrès des instructions, les différentes variantes du pipeline sont prouvées imperméables aux anomalies temporelles.

Encadrement de stages de Master

2023 Louison Jeanmougin, M2 : "Construction de CFGs au niveau warp pour analyse WCET des GPU", co-encadré avec Christine Rochange et Pascal Sotin

2023 Noïc Crouzet, M2 : "Design et synthèse de GPU open source pour la prédictibilité temporelle", co-encadré avec Christine Rochange

2022 Louison Jeanmougin, M1 : "Bibliothèque de modification de code ARM désassemblé"

2022 Célestin Grenier, 5e année ingénieur ISAE : "Implémentation du jeu d'instruction Kalray dans OTAWA"

2021 Michael Adalbert, M2 : "Modélisation du comportement temporel d'un GPU nvidia", co-encadré avec Christine Rochange

2021 Alban Gruin, M2 : "Modification d'un processeur Risc-V pour le rendre prédictible temporellement", co-encadré avec Christine Rochange

2020 Emmanuel Caussé, M2 : "Extraction de TIPs pour analyse statique d'applications critiques embarquées"

2020 Alexis Cornard, M2 : "Modélisation du jeu d'instruction des GPUs Nvidia Pascal en simnml", co-encadré avec Hugues Cassé

2020 Anthony Barrusseaud, M2 : "Apprentissage automatique de reconnaissance de feux de circulation ferroviaire et génération du code d'inférence embarqué correspondant en C", co-encadré avec Hugues Cassé

Participation à des Jurys de thèse

2023 — Nicolas Bellec, "Security enhancement in embedded hard real-time systems", (IRISA - Univ. Rennes 1)
Examinateur

Présentations invitées

2023 — **Journée commune GDR SOC2 – IRT saint Exupery : Scientific day on embedded high performance computing**, Toulouse
Timing predictability of GPUs: challenges and advances

2021 — **Université d'été École Temps Réel (ETR)**, Poitiers
Analyse statique de pire-temps d'exécution

Responsabilités collectives

2024 - — **Élu CNU** – Section 27

2024 - — **Élu de la Commission Formation et Vie Étudiante (CFVU) et du Conseil Académique (CAc)** de l'université Toulouse 3

2024 — **Workshop Chair** du workshop WCET

2024 — **Membre du comité de programme** de la conférence RTNS 2024

2023 — **Membre du comité local d'organisation** de la conférence HiPEAC 2023

2023 — **Organisateur** du workshop CAPITAL

2023 - — **Membre du comité de programme** de la conférence ECRTS

2022 — **Session chair** au workshop WCET

2022 - — **Membre du comité d'organisation** du workshop CAPITAL

2022 - — **Membre du comité de programme** du track "Work in Progress" de la conférence EMSOFT

2022 - — **Membre de l'équipe d'animation** de l'axe "Calcul embarqué haute performance" du GDR SOC2

2022 - — **Co-responsable** du domaine d'actions stratégiques "Aéronautique Espace Transports" de l'IRIT

2020 - — **Membre du comité de programme** de la conférence ARCS

2020 - 2023 — **Élu de la Commission Recherche et du Conseil Académique** de l'université Toulouse 3

2018 - — **Élu du collège scientifique** section 27 (et du GAEC B depuis 2022)

Publications

Journals

[6] Z. Bai, H. Cassé, T. Carle, and C. Rochange. "Computing Execution Times With Execution Decision Diagrams in the Presence of Out-of-Order Resources". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 42.11 (2023), pp. 3665–3678.

[7] Z. Bai, H. Cassé, M. De Michiel, T. Carle, and C. Rochange. "A Framework for Calculating WCET Based on Execution Decision Diagrams". In: *ACM Trans. Embed. Comput. Syst.* 21.3 (2022), 26:1–26:26.

[18] T. Carle and D. Potop-Butucaru. "Predicate-aware, makespan-preserving software pipelining of scheduling tables". In: *ACM Trans. Archit. Code Optim.* 11.1 (2014), 12:1–12:26.

[19] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens. "From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation". In: *Leibniz Trans. Embed. Syst.* 2.2 (2015), 01:1–01:30. DOI: 10.4230/LITES-v002-i002-a001.

[23] I. De Albuquerque Silva, T. Carle, A. Gauffriau, and C. Pagetti. "Extending a predictable machine learning framework with efficient gemm-based convolution routines". In: *Real Time Syst.* 59.3 (2023), pp. 408–437.

[34] A. Gruin, T. Carle, C. Rochange, H. Cassé, and P. Sainrat. "MINOTAuR: A Timing Predictable RISC-V Core Featuring Speculative Execution". In: *IEEE Transactions on Computers* 72.1 (2022), pp. 183–195. DOI: 10.1109/TC.2022.3200000.

Conferences and international workshops

[8] Z. Bai, H. Cassé, M. De Michiel, T. Carle, and C. Rochange. "Improving the Performance of WCET Analysis in the Presence of Variable Latencies". In: *Proceedings of the 21st ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2020, London, UK, June 16, 2020*. 2020.

[13] T. Carle and H. Cassé. "Reducing Timing Interferences in Real-Time Applications Running on Multicore Architectures". In: *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*.

[14] T. Carle and H. Cassé. "Static Extraction of Memory Access Profiles for Multi-core Interference Analysis of Real-Time Tasks". In: *Architecture of Computing Systems - 34th International Conference, ARCS 2021, Virtual Event, June 7-8, 2021, Proceedings*.

[15] T. Carle, M. Djemal, D. Genius, F. Pêcheux, D. Potop-Butucaru, R. de Simone, F. Wajsbürt, and Z. Zhang. "Reconciling performance and predictability on a many-core through off-line mapping". In: *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip, ReCoSoC 2014, Montpellier, France, May 26-28, 2014*. 2014.

[16] T. Carle, M. Djemal, D. Potop-Butucaru, R. de Simone, and Z. Zhang. "Static Mapping of Real-Time Applications onto Massively Parallel Processor Arrays". In: *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*. 2014.

[17] T. Carle, D. Papagiannopoulou, T. Moreshet, A. Marongiu, M. Herlihy, and R. I. Bahar. "Thrifty-malloc: A HW/SW codesign for the dynamic management of hardware transactional memory in embedded multicore systems". In: *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. 2016.

[21] I. De Albuquerque Silva, T. Carle, A. Gauffriau, V. Jegu, and C. Pagetti. "A predictable SIMD library for GEMM routines". In: *30th Real-Time and Embedded Technology and Applications Symposium, RTAS 2024, May 13-16, 2024, Hong Kong, China*. 2024.

[22] I. De Albuquerque Silva, T. Carle, A. Gauffriau, and C. Pagetti. "ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical Systems". In: *34th Euromicro Conference on Real-Time Systems, ECRTS 2022, July 5-8, 2022, Modena, Italy*.

[31] R. Gorcitz, E. Kofman, T. Carle, D. Potop-Butucaru, and R. de Simone. "On the Scalability of Constraint Solving for Static/Off-Line Real-Time Scheduling". In: *Formal Modeling and Analysis of Timed Systems - 13th International Conference, FORMATS 2015, Madrid, Spain, September 2-4, 2015, Proceedings*. Lecture Notes in Computer Science. 2015.

[33] A. Gruin, T. Carle, H. Cassé, and C. Rochange. "Speculative Execution and Timing Predictability in an Open Source RISC-V Core". In: *IEEE Real-Time Systems Symposium (RTSS)*. 2021, pp. 393–404. DOI: 10.1109/RTSS52674.2021.00043.

[35] A. Gruin, T. Carle, C. Rochange, and P. Sainrat. "Enabling timing predictability in the presence of store buffers". In: *31st International Conference on Real-Time Networks and Systems, RTNS 2023*. 2023.

[36] A. Gruin, T. Carle, C. Rochange, and P. Sainrat. "Validation of Processor Timing Models Using Cycle-Accurate Timing Simulators". In: *21th International Workshop on Worst-Case Execution Time Analysis, WCET 2023*. 2023.

[48] L. Jeanmougin, P. Sotin, C. Rochange, and T. Carle. "Warp-Level CFG Construction for GPU Kernel WCET Analysis". In: *21th International Workshop on Worst-Case Execution Time Analysis, WCET 2023, July 11, 2023, Vienna, Austria*.

[56] R. Meunier, T. Carle, and T. Monteil. "Correctness and Efficiency Criteria for the Multi-Phase Task Model". In: *34th Euromicro Conference on Real-Time Systems, ECRTS 2022, July 5-8, 2022, Modena, Italy*. 2022.

Bibliography

[1] absInt. *absInt aiT*. https://www.absint.com/ait/index.htm.

[2] S. Altmeyer and C. Maiza. "Cache-related preemption delay via useful cache blocks: Survey and redefinition". In: *Journal of Syst. Arch. - Embedded Systems Design* ().

[3] J. Arora, C. Maia, S. Aftab Rashid, G. Nelissen, and E. Tovar. "Bus-Contention Aware Schedulability Analysis for the 3-Phase Task Model with Partitioned Scheduling". In: *ACM International Conference Proceeding Series*. Association for Computing Machinery (ACM), 2021. DOI: 10.1145/3453417.3453433.

[4] M. Asavoae, B. Ben Hedia, and M. Jan. "Formal executable models for automatic detection of timing anomalies". In: *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[5] P. Axer et al. "Building Timing Predictable Embedded Systems". In: *ACM Transactions on Embedded Computing Systems* (2014).

[6] Z. Bai, H. Cassé, T. Carle, and C. Rochange. "Computing Execution Times With Execution Decision Diagrams in the Presence of Out-of-Order Resources". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 42.11 (2023), pp. 3665–3678.

[7] Z. Bai, H. Cassé, M. De Michiel, T. Carle, and C. Rochange. "A Framework for Calculating WCET Based on Execution Decision Diagrams". In: *ACM Trans. Embed. Comput. Syst.* 21.3 (2022), 26:1–26:26.

[8] Z. Bai, H. Cassé, M. De Michiel, T. Carle, and C. Rochange. "Improving the Performance of WCET Analysis in the Presence of Variable Latencies". In: *Proceedings of the 21st ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2020, London, UK, June 16, 2020*. 2020.

[9] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. "OTAWA: an Open Toolbox for Adaptive WCET Analysis (regular paper)". In: *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*. 2010.

[10] R. Bhargava and L.K. John. "Issues in the design of store buffers in dynamically scheduled processors". In: *2000 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS*. 2000, pp. 76–87. DOI: 10.1109/ISPASS.2000.842285.

[11] B. Binder, M. Asavoae, F. Brandner, B. Ben Hedia, and M Jan. "Formal modeling and verification for amplification timing anomalies in the superscalar TriCore architecture". In: *Int. J. Softw. Tools Technol. Transf.* 24.3 (2022), pp. 415–440. DOI: 10.1007/s10009-022-00655-1.

[12] B. Binder, M. Asavoae, F. Brandner, B. Ben Hedia, and M. Jan. "The Role of Causality in a Formal Definition of Timing Anomalies". In: *28th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2022, Taipei, Taiwan, August 23-25, 2022*. 2022, pp. 91–102. DOI: 10.1109/RTCSA55878.2022.00016.

[13] T. Carle and H. Cassé. "Reducing Timing Interferences in Real-Time Applications Running on Multicore Architectures". In: *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*.

[14] T. Carle and H. Cassé. "Static Extraction of Memory Access Profiles for Multi-core Interference Analysis of Real-Time Tasks". In: *Architecture of Computing Systems - 34th International Conference, ARCS 2021, Virtual Event, June 7-8, 2021, Proceedings*.

[15] T. Carle, M. Djemal, D. Genius, F. Pêcheux, D. Potop-Butucaru, R. de Simone, F. Wajsbürt, and Z. Zhang. "Reconciling performance and predictability on a many-core through off-line mapping". In: *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip, ReCoSoC 2014, Montpellier, France, May 26-28, 2014*. 2014.

[16] T. Carle, M. Djemal, D. Potop-Butucaru, R. de Simone, and Z. Zhang. "Static Mapping of Real-Time Applications onto Massively Parallel Processor Arrays". In: *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*. 2014.

[17] T. Carle, D. Papagiannopoulou, T. Moreshet, A. Marongiu, M. Herlihy, and R. I. Bahar. "Thrifty-malloc: A HW/SW codesign for the dynamic management of hardware transactional memory in embedded multicore systems". In: *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. 2016.

[18] T. Carle and D. Potop-Butucaru. "Predicate-aware, makespan-preserving software pipelining of scheduling tables". In: *ACM Trans. Archit. Code Optim.* 11.1 (2014), 12:1–12:26.

[19] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens. "From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation". In: *Leibniz Trans. Embed. Syst.* 2.2 (2015), 01:1–01:30. DOI: 10.4230/LITES-v002-i002-a001.

[20] R. I. Davis, S. Altmeyer, L. Indrusiak, C. Maiza, V. Nélis, and J. Reineke. "An extensible framework for multicore response time analysis". In: *Real Time Syst.* (2018).

[21] I. De Albuquerque Silva, T. Carle, A. Gauffriau, V. Jegu, and C. Pagetti. "A predictable SIMD library for GEMM routines". In: *30th Real-Time and Embedded Technology and Applications Symposium, RTAS 2024, May 13-16, 2024, Hong Kong, China*. 2024.

[22] I. De Albuquerque Silva, T. Carle, A. Gauffriau, and C. Pagetti. "ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical Systems". In: *34th Euromicro Conference on Real-Time Systems, ECRTS 2022, July 5-8, 2022, Modena, Italy*.

[23] I. De Albuquerque Silva, T. Carle, A. Gauffriau, and C. Pagetti. "Extending a predictable machine learning framework with efficient gemm-based convolution routines". In: *Real Time Syst.* 59.3 (2023), pp. 408–437.

[24] Théo Degioanni and Isabelle Puaut. "StAMP: Static Analysis of Memory Access Profiles for Real-Time Tasks". In: *20th International Workshop on Worst-Case Execution Time Analysis, WCET 2022, July 5, 2022, Modena, Italy*. 2022.

[25] B. Dupont de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. "Time-critical computing on a single-chip massively parallel processor". In: *Design, Automation and Test in Europe (DATE)*. 2014.

[26] M. Dupont de Dinechin, M. Schuh, M. Moy, and C. Maiza. "Scaling Up the Memory Interference Analysis for Hard Real-Time Many-Core Systems". In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020.

[27] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch. "Predictable Flight Management System Implementation on a Multicore Processor". In: *ERTS'14*. 2014.

[28] J. Eisinger, I. Polian, B. Becker, S. Thesing, R. Wilhelm, and A. Metzner. "Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis". In: *IEEE Design and Diagnostics of Electronic Circuits and systems*. 2006, pp. 13–18.

[29] S. Garcia-Esteban, A. Serrano-Cases, J. Abella, E. Mezzetti, and F. J. Cazorla. "Quasi Isolation QoS Setups to Control MPSoC Contention in Integrated Software Architectures". In: *35th Euromicro Conference on Real-Time Systems, ECRTS 2023, July 11-14, 2023, Vienna, Austria*. 2023.

[30] G. Gebhard. "Timing anomalies reloaded". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[31] R. Gorcitz, E. Kofman, T. Carle, D. Potop-Butucaru, and R. de Simone. "On the Scalability of Constraint Solving for Static/Off-Line Real-Time Scheduling". In: *Formal Modeling and Analysis of Timed Systems - 13th International Conference, FORMATS 2015, Madrid, Spain, September 2-4, 2015, Proceedings*. Lecture Notes in Computer Science. 2015.

[32] A. Gruin, T. Carle, H. Cassé, and C. Rochange. *Repository for our changes to MINOTAuR*. 2023. URL: `https://gitlab.irit.fr/minotaur/MINOTAuR`.

[33] A. Gruin, T. Carle, H. Cassé, and C. Rochange. "Speculative Execution and Timing Predictability in an Open Source RISC-V Core". In: *IEEE Real-Time Systems Symposium (RTSS)*. 2021, pp. 393–404. DOI: `10.1109/RTSS52674.2021.00043`.

[34] A. Gruin, T. Carle, C. Rochange, H. Cassé, and P. Sainrat. "MINOTAuR: A Timing Predictable RISC-V Core Featuring Speculative Execution". In: *IEEE Transactions on Computers* 72.1 (2022), pp. 183–195. DOI: `10.1109/TC.2022.3200000`.

[35] A. Gruin, T. Carle, C. Rochange, and P. Sainrat. "Enabling timing predictability in the presence of store buffers". In: *31st International Conference on Real-Time Networks and Systems, RTNS 2023*. 2023.

[36] A. Gruin, T. Carle, C. Rochange, and P. Sainrat. "Validation of Processor Timing Models Using Cycle-Accurate Timing Simulators". In: *21th International Workshop on Worst-Case Execution Time Analysis, WCET 2023*. 2023.

[37] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2022. URL: `https://www.gurobi.com`.

[38] H. 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 fuer Informatik, 2016, 2:1–2:10. ISBN: 978-3-95977-025-5. DOI: `10.4230/OASIcs.WCET.2016.2`. URL: `http://drops.dagstuhl.de/opus/volltexte/2016/6895`.

[39] S. Hahn, M. Jacobs, and J. Reineke. "Enabling Compositionality for Multicore Timing Analysis". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS '16*. 2016.

[40] S. Hahn and J. Reineke. "Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core". In: *IEEE Real-Time Systems Symposium (RTSS)*. 2018, pp. 469–481. DOI: `10.1109/RTSS.2018.00060`.

[41] S. Hahn and J. Reineke. "Design and analysis of SIC: A provably timing-predictable pipelined processor core". In: *Real Time Systems* 56.2 (2020), pp. 207–245. DOI: 10.1007/s11241-019-09341-z.

[42] S. Hahn, J. Reineke, and R. Wilhelm. "Toward compact abstractions for processor pipelines". In: *Correct System Design*. Springer, 2015, pp. 205–220.

[43] S. Hahn, J. Reineke, and R. Wilhelm. "Towards compositionality in execution time analysis: definition and challenges". In: *ACM SIGBED Review* 12.1 (2015), pp. 28–36. DOI: 10.1145/2752801.2752805.

[44] Sebastian Hahn. "On Static Execution-time Analysis: Compositionality, Pipeline Abstraction, and Predictable Hardware". PhD thesis. Universität des Saarlandes, 2018.

[45] Z. Hanzálek and P. Šůcha. "Time symmetry of resource constrained project scheduling with general temporal constraints and take-give resources". In: *Annals of Operations Research* 248.1-2 (Jan. 2017), pp. 209–237. ISSN: 15729338. DOI: 10.1007/S10479-016-2184-6/TABLES/4.

[46] J. Hennessy and D. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011.

[47] M. Jan, M. Asavoae, M. Schoeberl, and E. A. Lee. "Formal Semantics of Predictable Pipelines: a Comparative Study". In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2020, pp. 103–108. DOI: 10.1109/ASP-DAC47756.2020.9045351.

[48] L. Jeanmougin, P. Sotin, C. Rochange, and T. Carle. "Warp-Level CFG Construction for GPU Kernel WCET Analysis". In: *21th International Workshop on Worst-Case Execution Time Analysis, WCET 2023, July 11, 2023, Vienna, Austria*.

[49] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System". In: *The Computer Journal* (1986).

[50] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance". In: *30th IEEE International Conference on Computer Design (ICCD)*. 2012.

[51] I. Liu, J. Reineke, and E. A. Lee. "A PRET architecture supporting concurrent programs with composable timing properties". In: *Asilomar Conference on Signals, Systems and Computers*. 2010.

[52] T. Lundqvist and P. Stenstrom. "Timing anomalies in dynamically scheduled microprocessors". In: *IEEE Real-Time Systems Symposium*. IEEE. 1999, pp. 12–21. DOI: 10.1109/REAL.1999.818824.

[53] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi. "A Survey on Static Cache Analysis for Real-Time Systems". In: *Leibniz Transactions on Embedded Systems* 3.1 (2016).

[54] C. Maiza, H. Rihani, J. H. Rivas, J. Goossens, S. Altmeyer, and R.I. Davis. "A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems". In: *ACM Comp. Surv.* (2019).

[55] J. Matějka, B. Forsberg, M. Sojka, P. Šůcha, L. Benini, A. Marongiu, and Z. Hanzálek. "Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution". In: *Parallel Computing* 85 (July 2019), pp. 27–44.

[56] R. Meunier, T. Carle, and T. Monteil. "Correctness and Efficiency Criteria for the Multi-Phase Task Model". In: *34th Euromicro Conference on Real-Time Systems, ECRTS 2022, July 5-8, 2022, Modena, Italy*. 2022.

[57] R. Meunier, T. Carle, and T. Monteil. "Correctness and Efficiency Criteria for the Multi-Phase Task Model". In: *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Ed. by Martina Maggio. Vol. 231. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 9:1–9:21. ISBN: 978-3-95977-239-6. DOI: 10.4230/LIPIcs.ECRTS.2022.9.

[58] T. Mitra. "Time-Predictable Computing by Design: Looking Back, Looking Forward". In: *Annual Design Automation Conference*. 2019.

[59] T. Mitra, J. Teich, and L. Thiele. "Time-Critical Systems Design: A Survey". In: *IEEE Design and Test* 35.2 (2018).

[60] F. Nemer, H. Cassé, P. Sainrat, J-P. Bahsoun, and M. De Michiel. "PapaBench: a Free Real-Time Benchmark". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)* (2006). DOI: 10.4230/OASICS.WCET.2006.678.

[61] C. Pagetti, J. Forget, H. Falk, D. Oehlert, and A. Luppold. "Automated Generation of Time-Predictable Executables on Multicore". In: *Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS '18)*. 2018.

[62] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. "The ROSACE case study: From Simulink specification to multi/many-core execution". In: *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*. IEEE Computer Society, 2014, pp. 309–318. DOI: 10.1109/RTAS.2014.6926012.

[63] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. "A Predictable Execution Model for COTS-Based Embedded Systems". In: *RTAS*. 2011.

[64] R. Pellizzoni, A. Schranzhofer, M. Caccamo, and L. Thiele. "Worst case delay analysis for memory interference in multicore systems". In: *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*. 2010, pp. 741–746.

[65] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. "A Definition and Classification of Timing Anomalies". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Ed. by Frank Mueller. Vol. 4. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. ISBN: 978-3-939897-03-3. DOI: 10.4230/OASIcs.WCET.2006.671. URL: http://drops.dagstuhl.de/opus/volltexte/2006/671.

[66] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut. "Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures". In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. 2019.

[67] O. Sander, F. Bapp, L. Dieudonne, T. Sandmann, and J. Becker. "The promised future of multi-core processors in avionics systems". In: *CEAS Aeronautical Journal* (2017). ISSN: 18695590. DOI: 10.1007/s13272-016-0228-x.

[68] J. Schneider, M. Bohn, and R. Rößger. "Migration of automotive real-time software to multicore systems: First steps towards an automated solution". In: *22nd EUROMICRO Conference on Real-Time Systems*. 2010.

[69] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, and C. W. Probst. "Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach". In: *Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. 2011.

[70] M. Schoeberl et al. "T-CREST: Time-predictable multi-core architecture for embedded systems". In: *Journal of Systems Architecture* (2015).

[71] M. Schuh, C. Maiza, J. Goossens, P. Raymond, and B. Dupont de Dinechin. "A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory". In: *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*. IEEE, 2020, pp. 283–295. DOI: 10.1109/RTSS49844.2020.00034.

[72] ThalesGroup. *CVA6-softcore-contest*. URL: https://github.com/ThalesGroup/cva6-softcore-contest/tree/0abb1a6.

[73] L. Thiele, S. Chakraborty, and M. Naedele. "Real-Time Calculus For Scheduling Hard Real-Time Systems". In: *in ISCAS*. 2000.

[74] Field G. Van Zee and Robert A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Transactions on Mathematical Software* 41.3 (2015), 14:1–14:33. ISSN: 0098-3500. DOI: 10.1145/2764454. URL: https://doi.org/10.1145/2764454 (visited on 10/15/2022).

[75] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. "Principles of timing anomalies in superscalar processors". In: *Fifth International Conference on Quality Software (QSIC'05)*. 2005, pp. 295–303. DOI: 10.1109/QSIC.2005.49.

[76] Zhang Xianyi, Wang Qian, and Werner Saar. *OpenBLAS: An optimized BLAS library*. 2011. URL: https://www.openblas.net/.

[77] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. "Invisispec: Making speculative execution invisible in the cache hierarchy". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 428–441.

[78] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2019).

[79] M. Zini, D. Casini, and A. Biondi. "Analyzing Arm's MPAM From the Perspective of Time Predictability". In: *IEEE Trans. Computers* 72.1 (2023), pp. 168–182.