MINOTAuR: a Timing Predictable RISC-V Core Featuring Speculative Execution

Alban Gruin, Thomas Carle, Christine Rochange, Hugues Cassé, Pascal Sainrat

MINOTAuR: a Timing Predictable RISC-V Core Featuring Speculative Execution

Alban Gruin, Thomas Carle, Christine Rochange, Hugues Cassé and Pascal Sainrat
IRIT - Univ. Toulouse 3 - CNRS
Toulouse, France
name.surname@irit.fr

Abstract—
We present MINOTAuR, an open-source RISC-V core designed to be timing predictable, i.e. free of timing anomalies: this property enables a compositional timing analysis in a multicore context. MINOTAuR features speculative execution: thanks to a specific design of its pipeline, we formally prove that speculation does not break timing predictability while sensibly increasing performance. We propose architectural extensions that enable the use of a return address stack and of any cache replacement policy, which we implemented in the MINOTAuR core. We show that a trade-off can be found between the efficiency of these components and the overhead they incur on the die area consumption, and that using them yields a performance equivalent to that of the baseline RISC-V Ariane core, while also enforcing timing predictability.

Index Terms—timing predictability, processor architecture.

1 Introduction

The ever-growing performance requirements of embedded real-time systems lead to implement them on multicore 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 [9] 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* [21]: a local best case situation (e.g. a cache hit) does not necessarily lead to the global worst case (that is the worst-case execution time of the task under analysis). Timing anomalies are mainly related to possible 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 [10] 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.

Our objective in this paper is to leverage this approach and its formal framework to a more complex core with a higher baseline performance than the one used in [10]: the open source RISC-V Ariane [27] core, which implements the

This work was partially funded by the French National Research Agency (ANR) under the "Investissements d'Avenir" ANR-11-LABX-0040 reference.

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.

This paper is an extension of [8]. The key differences are the following:

- we present a modification of the Ariane core that makes it faster. It also reduces the overhead between Ariane and MINOTAuR. The modification is introduced in Section 3.1, highlighted in Figure 1 and taken into account in the model of Figure 3.
- we introduce novel mechanisms to handle LRU caches and a Return Address Stack during speculative execution (Section 4.3). We also have synthesized our processor onto an FPGA and present measured instead of simulated performance. This also allows us to provide information on LUT usage and maximum achievable frequency for our core (Section 5).

The rest of the paper is organized as follows. Section 2 presents related work on timing predictable processors and introduces the SIC approach in more details. Section 3 describes the architecture of the Ariane core and shows an example of a timing anomaly. In Section 4, we introduce the MINOTAuR core, prove its timing predictability, and present hardware extensions for caches and return address stacks. We evaluate our design in Section 5. Finally, Section 6 concludes the paper and presents some perspectives.

2 Related work

2.1 Timing predictability

A processor is said *timing predictable* when there are no timing anomalies and it is timing compositional [10].

A timing anomaly is a situation where a local worst case (e.g. conservatively considering a cache access as a miss) does not lead to the global worst case (i.e. the execution time with that assumption is not the longest one) [17]. 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 [1], [4], [6], [21], [25]. 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 2.2).

Timing compositionality simplifies the timing analysis of a multi-core system [13]. 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 [9].

2.2 Timing predictable processor architectures

Several ways have been considered to favor timing predictability in hardware platforms [2], [19], [20].

The Kalray MPPA-256 processor [3] 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 [15] is an implementation of a precision-timed (PRET) machine [16]. 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 [23] 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 [22]. Although it has been designed to be timing predictable, this has not been formally proven to the best of our knowledge.

In [10], [11], 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 2.3. 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.

2.3 A formal framework to prove timing predictability

A framework to express the concrete semantics of a processor pipeline is proposed in [12]. 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 $\mathcal{P}$:

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

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 :

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

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 [10], 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 [10].

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 [10] to any in-order pipeline that enforces Property 1. The proof arguments of [10] 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 [10] demonstrate that the SIC processor is free of timing anomalies with respect to uncertain cache behaviour, and timing-compositional with respect to uncertain 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.*

We reformulate the proof of this theorem here to make it more general.

Proof. Let c be the state that splits upon the cache uncertainty of instruction i, leading to the hit-case successor state c_b and miss-case successor c_w. Without loss of generality, we consider a data cache miss. We need to show that $c_w \sqsubseteq c_b$, which, with Theorem 2, proves Theorem 3.

- Due to Lemma 1, the progress of instructions $j < i$ does not depend on the uncertainty of instruction i, so $c_b(j) = c_w(j)$.
- For instruction i, we know that $c_w(i) \sqsubseteq_{\mathcal{P}} c_b(i)$. In practice, if s is the pipeline stage where the access to the cache is performed, $c_b(i) = (s, lat_{hit})$ and $c_w(i) \sqsubseteq_{\mathcal{P}} (s, lat_{miss})$ with $lat_{hit} < lat_{miss}$. Note that $c_w.stg(i) \sqsubset_{\mathcal{S}} s$ is possible if accessing the memory to load data into the cache upon a miss is stalled by an older instruction, e.g. a $store$.
- For instructions $k > i$, $c_w(k) \sqsubseteq_{\mathcal{P}} c_b(k)$ follows from the fact that $c_b.ready(k)$ is true if $c_w.ready(k)$ is true. Thus if k has progressed in c_w, it has progressed in c_b as well.

□

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 [10].

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 [10] is specific to the SIC processor.

2.4 Hardware state buffering mechanisms

In Section 4.3 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 [14] but, to the best of our knowledge, they were not implemented. InvisiSpec [26] 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,

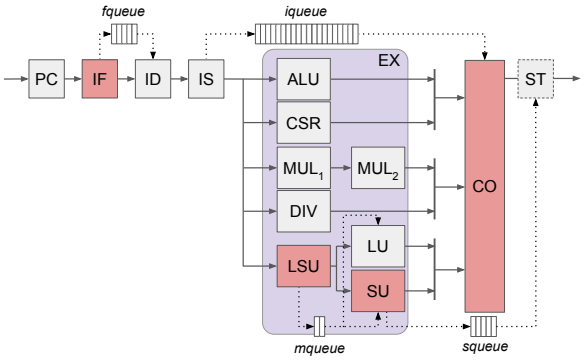


Fig. 1: Model of the Ariane$^+$ core pipeline.

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.

3 A BASELINE RISC-V CORE

Our baseline core is a slightly modified version of the Ariane core [27], a 6-stage in-order RISC-V processor.

3.1 The original Ariane architecture

The structure of the Ariane core is depicted in Figure 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 whereas the SU keeps instructions in a 4-slot store buffer. 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.
- ALU, multiplier/divider and CSR instructions cannot be dispatched as long as a CSR instruction is pending
- the SU cannot accept any instruction as long as the AMO buffer is not empty. The LU cannot accept any instruction as long as the AMO and store buffers are 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.

The baseline version of Ariane that we use implements the RV32IMAC instruction set [24]. 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 remainder of the paper, we call Ariane$^+$ the version of the core that includes these modifications.

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) [13]. We refer to this phenomenon as an *inversion*.

We illustrate how inversions can lead to timing anomalies in Figure 2. This figure displays the execution timing of a simple sequence composed of 6 instructions in the pipeline of Ariane$^+$. We added a retire (RE) stage in order to show

1. Even though the divider has a variable latency.

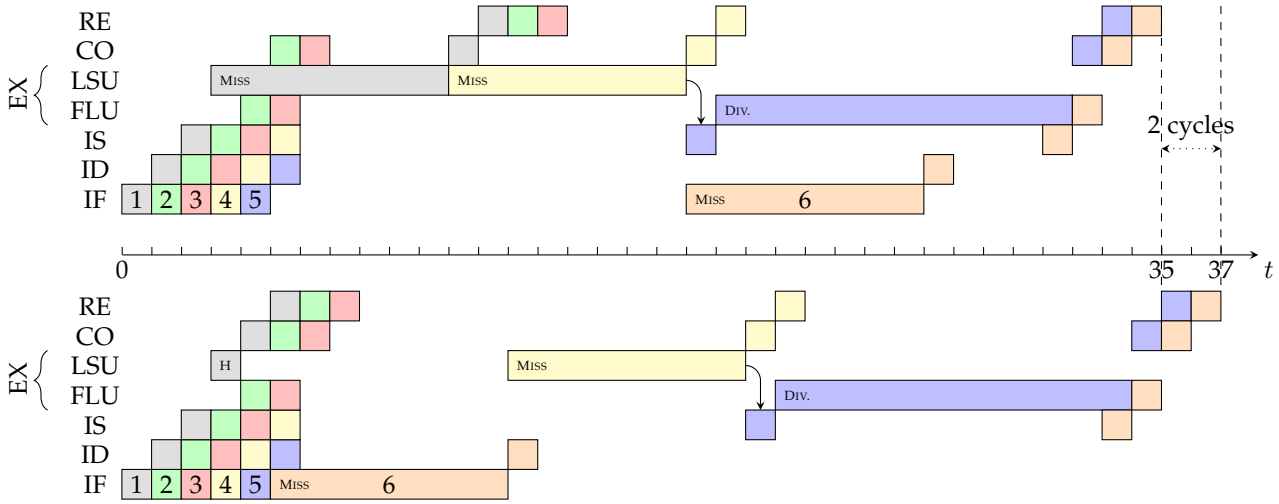


Fig. 2: Example of a timing anomaly on the Ariane processor.

that the instructions are retired in order. Instructions 1 and 4 are memory loads, instruction 5 is a division and a data dependency exists between instructions 4 and 5 (one of the operands of the division is loaded by instruction 4). The rest of the instructions (2, 3 and 6) are integer additions. We assume that instruction 6 leads to a cache miss in the FE stage and that instruction 4 leads to a cache miss in the MEM stage.

At the top of Figure 2, we display the execution timing of the sequence if instruction 1 leads to a miss in the data cache (in the MEM stage). This miss blocks the fetch of instruction 6 (because instruction 6 leads to a miss in the instruction cache). When instruction 1 leaves the MEM stage, instruction 4 enters it and also produces a miss in the data cache. This miss further postpones the fetch of instruction 6, because in case of simultaneous access, the data cache has priority over the instruction cache on their shared bus.

At the bottom of Figure 2, we display the execution of the same instruction sequence when instruction 1 leads to a hit in the data cache. Instruction 6 leads to a miss in the instruction cache at cycle 5. Then, instruction 4 must wait for instruction 6 to free the bus before it can enter the memory stage: this is an inversion. Since instruction 5 depends on the value loaded by instruction 4, it cannot enter the issue stage until cycle 21.

This example shows a timing anomaly in the Ariane core: a data cache miss for instruction 1 leads to an execution time of 35 cycles for the sequence, while a cache hit for the same instruction leads to an execution time of 37 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 5.1. 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.

4 MINOTAuR: A TIMING PREDICTABLE CORE

4.1 Enforcing timing predictability

The MINOTAuR 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.

We start by providing the formal model of MINOTAuR, and then prove its timing predictability in the presence of speculative execution.

4.2 Formal model and proofs

4.2.1 Definitions

Each instruction $i \in \mathcal{I}$ is characterized by its category $opc(i) \in \{branch, store, load, atomic, mul, div, csr\}$ 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 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

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

- $c.nstg(i)$: 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 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 ID to IS (the instruction is stalled if a *csr* instruction is pending, or if a dependency exists with a previous instruction – modelled by the $dep(i_1, i_2)$ predicate – which has not reached 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.

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 1.

4.2.2 Timing predictable speculative execution

As pointed out in Section 3, MINOTAuR features a branch predictor which 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.

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 will 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 [18] 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 will then describe and evaluate general mechanisms that can be added to any RAS or cache in order to lift these restrictions.

Additionally, as speculated store instructions cannot perform their write to memory (in stage ST, i.e. after stage CO)

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3: Model of the MINOTAuR core.

before the corresponding branch instruction is resolved, we do not need to consider the effect of stores in our proofs.

4.2.3 Timing anomaly freedom proofs

We start by proving 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:

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

Theorem 6 (Absence of cache state modification during speculation)**.** $\forall i \in \mathcal{I}, \forall c \in \mathcal{C}, c.pending(i, branch) \Rightarrow \neg cycle(c).cmod(i)$

Proof. Let $i \in \mathcal{I}$ and $c \in \mathcal{C}$. By definition, $cycle(c).cmod(i)$ is equivalent to:

$$(cycle(c).stg(i) = \text{IF} \wedge \neg ichit(i)) \quad (1)$$
$$\vee (cycle(c).stg(i) = \text{LU} \wedge \neg dchit(i)) \quad (2)$$

We will show that none of these terms hold.
Let us first assume that $c.stg(i) \sqsubset_{\mathcal{S}} \text{PC}$. Then trivially, $cycle(c).stg(i) \sqsubset_{\mathcal{S}} \text{IF}$. Let us now consider $c \in \mathcal{C}$ such that $c.stg(i) = \text{PC}$ and $c.pending(i, branch)$. Let us also assume that $\neg ichit(i)$ (otherwise (1) does not hold). Since $c.pending(i, branch) \wedge \neg ichit(i) \Rightarrow \neg c.ready(i)$, we deduce that $cycle(c).stg(i) = \text{PC}$. We can recursively apply the same argument to prove that for all states c' such that $c'.pending(i, branch)$, $cycle(c').stg(i) \sqsubset_{\mathcal{S}} \text{IF}$. From this we conclude that (1) does not hold.

Now let us consider again $i \in \mathcal{I}$ and $c \in \mathcal{C}$ such that $c.pending(i, branch)$. By definition of $pending$, we know that $\exists j_{br} < i \,.\, opc(j_{br}) = branch \wedge c(j_{br}) \sqsubset_{\mathcal{P}} (\text{ALU}, 0)$. Then, given the structure of the pipeline, we can deduce that $c.stg(i) \sqsubseteq_{\mathcal{S}} c.stg(j_{br}) \sqsubseteq_{\mathcal{S}} \text{IS}$. If $c.stg(j_{br}) \sqsubset_{\mathcal{S}} \text{IS}$, then trivially $cycle(c).stg(i) \sqsubset_{\mathcal{S}} \text{LSU}$ and (2) does not hold. If $c.stg(j_{br}) = \text{IS}$, then necessarily $c.stg(i) \sqsubset_{\mathcal{S}} \text{IS}$ and once again $cycle(c).stg(i) \sqsubset_{\mathcal{S}} \text{LSU}$, so (2) cannot hold. □

Since this proof does not make any assumption on the position of j_{br} in case of nested branch predictions, Theorem 6 remains valid for any instruction i as long as there exists an unresolved branch instruction that precedes i.

In this proof we showed that no memory access is performed speculatively: it results 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.

We now prove that MINOTAuR fulfills Property 1. We focus on the blue parts of the model since it is specific to speculation. We prove that the rest of the model fulfills Property 1 in the appendix.

Theorem 7 (Update enable in MINOTAuR)**.** *The MINOTAuR core satisfies Property 1.*

Proof. Let $c_a, c_b \in \mathcal{C}$ be two pipeline states and i be an instruction such that $c_a(i) = c_b(i) \wedge (\forall j < i, c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)))$. We must prove that:

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

We start with the *ready* case, focusing on the blue parts. The rest of the proof that does not concern speculative execution is given in the appendix. From $c_a(i) = c_b(i)$, we get $c_a.stg(i) \neq pre \Rightarrow c_b.stg(i) \neq pre$ and $c_a.stg(i) = \text{PC} \Rightarrow c_b.stg(i) = \text{PC}$. Moreover, since $\forall j < i,\ c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$, it follows that $\neg c_a.pending(i, op) \Rightarrow \neg c_b.pending(i, op)$. Finally, $pwrong(i)$ and $ichit(i)$ only depend on the instruction and not on the pipeline state. As a result, $c_a.ready(i) \Rightarrow c_b.ready(i)$.

The same arguments apply to the blue case in *free*, then $c_a.free(c_a.nstg(i)) \Rightarrow c_b.free(c_b.nstg(i))$. □

Using Theorem 7, we conclude 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) which 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:*

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

Proof. Instructions $j \leq i_{br}$ are all executed non speculatively and belong to both sequences $\mathcal{I}_1$ and $\mathcal{I}_2$. Since the pipeline satisfies Property 1, the progress of these instructions does not depend on the following instructions. As a result, $\forall j \leq i_{br}, c_w(j) = c_b(j)$, since c_w and c_b correspond to the same number of applications of $cycle$ since the beginning of the sequence.
By definition of state c_w, all speculated instructions have been flushed from the pipeline in this state. Thus in the rest of the proof, we will only consider the non-speculated instructions (i.e. we consider only $\mathcal{I}_1$).

Instructions $j > i_{br+1}$ have not yet entered the pipeline in state c_w: $\forall j > i_{br+1}, c_w.stg(j) = pre$. Thus, regardless of their progress in c_b, we have $c_w(j) \sqsubseteq_{\mathcal{P}} c_b(j)$.
Now, for i_{br+1}, we can write: $c_w.stg(i_{br+1}) \sqsubseteq_{\mathcal{S}} \text{PC}$, because c_w is the pipeline state just after branch i_{br} has been resolved. If $c_w.stg(i_{br+1}) = pre$, then $c_w(i_{br+1}) \sqsubseteq_{\mathcal{P}} c_b(i_{br+1})$ is trivial. If $c_w.stg(i_{br+1}) = \text{PC}$, then $\forall j < i_{br+1}, \text{PC} \sqsubset_{\mathcal{S}} c_w.stg(j)$. Once again, by the progress dependence property, we also have $\forall j < i_{br+1}, \text{PC} \sqsubset_{\mathcal{S}} c_b.stg(j)$, so no prior instruction resides in PC in state c_b. Since i_{br+1} was able to leave pre and enter PC in c_w, i_{br+1} must also have been able to enter PC at least in c_b if not in a prior state. We thus have $\text{PC} \sqsubseteq_{\mathcal{S}} c_b.stg(i_{br+1})$, and thus $c_w(i_{br+1}) \sqsubseteq_{\mathcal{P}} c_b(i_{br+1})$. □

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

proceed with the next theorem which bounds the timing penalty for a branch misprediction in MINOTAuR.

Theorem 9 (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.*

Proof. We use the same notations as for Theorem 8. We consider the state c'_w obtained by applying $cycle$ to state c_w until instruction i_{br+1} reaches the same progress than in c_b i.e. until we reach $c'_w(i_{br+1}) = c_b(i_{br+1})$. Without loss of generality, we consider that c'_w is reached from c_w by applying $cycle$ $k > 0$ times. We prove that (1) $k \leq p$ and (2) the time penalty induced by a misprediction is bounded by k.
(1) c_b is obtained from c_{br} by applying $cycle$ p times. Since the progress of instructions in the pipeline does not depend on subsequent instructions, and since the pipeline guarantees a strict progress, we can derive that $\forall j \leq i_{br}, c_{br}(j) \sqsubseteq_{\mathcal{P}} c_w(j)$. We thus have the guarantee to reach c'_w from c_w in at most p cycles: $k \leq p$.
(2) (a) We start by proving that $c_b \sqsubseteq c'_w$: $\forall j \leq i_{br}, c_w(j) = c_b(j)$ and $c_w(j) \sqsubseteq_{\mathcal{P}} c'_w(j)$, so $c_b(j) \sqsubseteq_{\mathcal{P}} c'_w(j)$. By definition, we also have $c'_w(i_{br+1}) = c_b(i_{br+1})$. We wish to show that $\forall j \,.\, j > i_{br+1}, c_b(j) \sqsubseteq_{\mathcal{P}} c'_w(j)$. We can proceed by induction on j. We just stated that all instructions $j' \leq i_{br+1}$ are at least as advanced in c'_w as in c_b. It follows that the progress of instruction j which just follows i_{br+1} is less or equally constrained in c'_w than in c_b: j cannot be blocked in c'_w if it is not blocked in c_b and thus $c_b(j) \sqsubseteq_{\mathcal{P}} c'_w(j)$. We can repeat this argument for any $j > i_{br+1}$ to conclude that $c_b \sqsubseteq c'_w$.
(b) By applying Theorem 2, we obtain that $\forall i \in \mathcal{I}_1, f(c'_w, i) \leq f(c_b, i)$, which means $\forall i \in \mathcal{I}_1, f(c_w, i) - k \leq f(c_b, i)$: we conclude that the penalty for mispredicting the branch is at most k cycles.
From (1) and (2) we conclude that the penalty for mispredicting the branch is bounded by p cycles.

□

4.3 Releasing the constraints on the RAS and caches

In Section 4.2.2, 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.

4.3.1 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 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 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. 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 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 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.

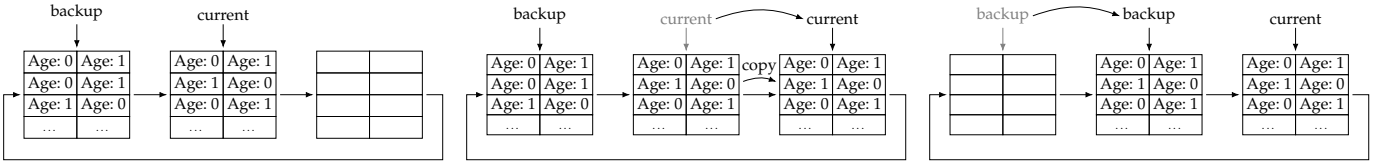
When the buffer that holds the copies is full (as the result of too many nested branches), new branch instructions are blocked before they can enter the IF stage to prevent any unsaved modification to the cache state. When a pending branch is resolved, the buffer recovers at least one slot, and branch 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 5.2.

4.3.2 RAS backup mechanism

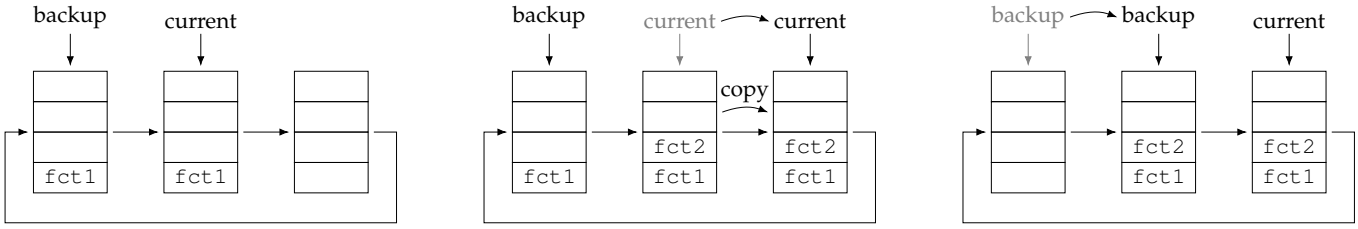
A return address stack (RAS) is a branch prediction mechanism used to predict the return address at the end of a function call: when a return instruction (e.g. `jr ra` in RISC-V) is executed at the end of function, the RAS predicts the address of the next instruction to execute. In a simple RAS, as found 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.



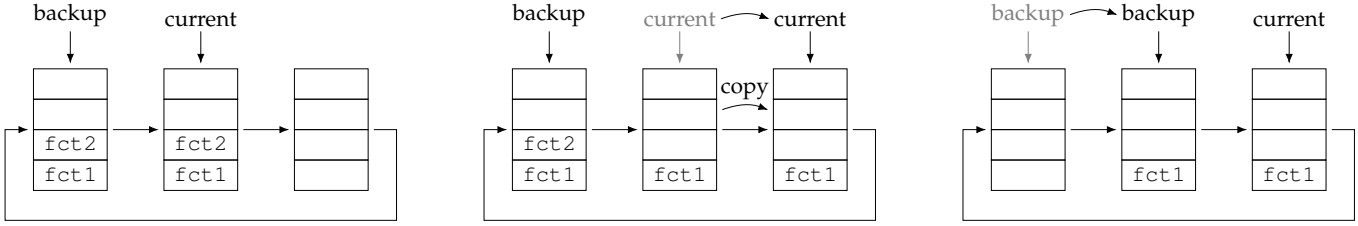
(a) During speculative execution (b) New branch prediction (c) Branch validation

Fig. 4: Cache state backup mechanism.



(a) During speculative execution (b) Call to function fct2 (c) Branch validation

Fig. 5: RAS state backup mechanism: function call.



(a) During speculative execution (b) Return from function fct2 (c) Branch validation

Fig. 6: RAS state backup mechanism: return from function.

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 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 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 6.

5 Experimental evaluation

5.1 Methodology

All our extensions have been implemented in the SystemVerilog model of the Ariane$^+$ core and our processor has been synthesized with Xilinx Vivado 2021.1[3], targeting

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

a Xilinx Zynq XC7Z020-1CLG400 on a Digilent Zybo Z7-20 board, with the `PerformanceOptimized` directive set[4], running at a frequency of 25 MHz. All the results reported in the paper come from real measurements. The memory has a latency of 11 cycles.

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

We started by comparing Ariane$^+$ to a baseline version of MINOTAuR, in which the RAS is disabled and the caches have a random replacement policy. The detailed results are provided in Table 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 Ariane$^+$ and MINOTAuR) and the difference between the geometric means of the number of cycles in Ariane$^+$ and MINOTAuR over the TACLe benchmarks.

Then, we enabled the RAS and replaced the random instruction cache by an LRU cache to evaluate the cost and benefit of our backup mechanisms. We then performed measurements while varying two parameters: the size of the RAS, and the depth of the backup mechanisms (the number of copies they can hold). We compared these measurements to the ones obtained with the version of MINOTAuR without RAS and with random caches. The results are reported in Table 2. In this table we provide the resource usage on the FPGA, the CoreMark score, the total number of cycles to run the TACLe benchmark suite, the average overhead, the global overhead and the standard deviation of each overhead.

The source code for all cores and experiments presented in this paper is available in [7].

TABLE 1: Results of the TACLe benchmark suite ran on Ariane$^+$ and on the base MINOTAuR.

	Ariane$^+$		MINOTAuR
Benchmark	Inversions	Cycles	Overhead
binarysearch	0	1,294	7.73%
bitcount	2	19,951	3.42%
bitonic	0	10,086	2.97%
bsort	0	60,087	0.11%
complex_updates	0	22,342	-1.14%
cosf	0	335,880	1.74%
countnegative	0	18,553	0.27%
cubic	1	13,135,490	2.20%
deg2rad	0	175,341	0.24%
fac	1	354	18.36%
fft	1	2,208,391	-0.68%
filterbank	0	48,117,007	1.79%
fir2dim	0	37,951	-2.06%
iir	0	6,343	5.09%
insertsort	0	1,374	20.31%
isqrt	0	525,191	0.02%
jfdctint	1	4,355	8.50%
lms	0	2,701,524	-3.29%
ludcmp	1	56,755	-2.67%
matrix1	0	11,839	1.04%
md5	0	7,690,282	-1.65%
minver	1	25,579	-1.11%
pm	0	121,648,002	1.10%
prime	1	618	21.04%
quicksort	0	4,497,647	1.43%
rad2deg	0	173,388	0.83%
recursion	1	2,669	11.46%
sha	1	932,360	-0.49%
st	0	1,973,611	0.06%
adpcm_dec	1	130,724	0.78%
adpcm_enc	1	132,760	0.65%
ammunition	1	287,442,316	0.64%
anagram	1	1,607,615	1.36%
audiobeam	0	4,070,266	1.41%
cjpeg_transupp	0	1,992,167	0.03%
cjpeg_wrbmp	0	77,943	0.43%
dijkstra	0	35,829,927	0.00%
epic	0	39,898,823	-0.20%
fmref	1	7,403,288	0.17%
g723_enc	0	564,614	-0.93%
gsm_dec	1	1,321,184	0.03%
gsm_enc	1	3,856,357	-0.03%
h264_dec	0	236,189	0.23%
huff_dec	1	116,986	0.97%
huff_enc	0	485,721	0.58%
ndes	0	59,279	1.45%
petrinet	1	1,305	-17.78%
rijndael_dec	1	4,893,874	-0.54%
rijndael_enc	0	4,660,858	0.08%
statemate	0	40,906	4.68%
Arithmetic mean			1.81%
Geometric mean			1.65%
Global overhead			0.69%

5.2 Results

The first observation is that we did not measure any inversion in the MINOTAuR core, 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. 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 consist on 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 refetched after a wrongly speculated branch.

Let us now focus on the results of Table 2. We first evaluated the efficiency of our backup mechanisms. To do so, we set the size of the RAS to 16 (thus allowing fast returns for up to 16 nested function calls), and varied

4. Except for 16 backups/RAS-16, which would not fit on our FPGA with this configuration. Synthesis for this model was made with the default parameters.

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

6. www.coremark.org

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

TABLE 2: Resource usage, CoreMark score and average overhead for Ariane+ and multiple MINOTAuR variants.

Core	LUTs	Max freq.	CoreMark score	Total cycles	Arith. mean	Geo. mean	Global overhead	Deviation
Ariane+	17,106	34.93 MHz	110.36	599,217,366				
MINOTAuR	17,176	32.97 MHz	110.58	603,373,808				
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 size of the RAS and LRU backups. We first see that using 2-slot buffers for the backup mechanism yield 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.

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.

6 Conclusion

In this paper we presented MINOTAuR, a timing predictable core based on the open source Ariane RISC-V core. We applied the principles of the SIC processor [10] and went further by allowing speculative execution with LRU caches. We showed that we could still formally prove timing predictability. The performance cost of these extensions compared to our baseline (non predictable) processor is around 1% on average, showing that timing predictability is compatible with performance-oriented mechanisms such as dynamic branch prediction, parallel functional units and caches with an aging-based replacement policy, and that timing predictable cores can achieve high performance. We provide the SystemVerilog source code of MINOTAuR and all intermediate designs presented in the paper in [7].

Limitations and future work

In this work we assumed a write-through data cache with a store buffer. We wish to generalize our results to any cache write policy, in more complex out-of-order pipelines. In this sense, further work is needed to assess the impact on the core predictability of using write-back caches that could speculatively fetch blocks for data writes. In the same fashion, our model of the store buffer and of its concurrency with the Load Unit is specific to MINOTAuR. More parallel implementations may break timing predictability, so we need to work on defining the minimal set of rules that guarantees the predictability of this mechanism, in the same way as we did for speculation.

In the close future, we plan to improve our backup mechanism to avoid unnecessary copies during unconditional jumps, function calls and returns, in order to reduce the size of MINOTAuR and improve its performance. We also plan to add a MMU to MINOTAuR while retaining predictability, run a real-time operating system that will allow us to evaluate our core on complete real-time applications, and perform static WCET analysis on the core.

We also envision the design and proof of a predictable superscalar out-of-order core. This work is much more ambitious and requires the extension of the formal modeling framework and a relaxation of the notion of monotonicity in order to accommodate out-of-order execution patterns that remain predictable.

7 Appendix

7.1 Remainder of proof of Theorem 7 for MINOTAuR

Let us consider two pipeline states $c_a, c_b \in \mathcal{C}$ and an instruction $i \in \mathcal{I}$ such that $c_a(i) = c_b(i)$. Let us assume that all previous instructions $j < i$ are such that $c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$. We first prove the following statements:

(a) $c_a.cnt(i) = 0 \Rightarrow c_b.cnt(i) = 0$
This follows from $c_a(i) = c_b(i)$.

(b) $c_a.nstg(i) = c_b.nstg(i)$
This follows from $c_a(i) = c_b(i)$.

(c) $c_a.isnext(i, c_a.stg(i)) \Rightarrow c_b.isnext(i, c_b.stg(i))$
- $c_a(i) = c_b(i) \Rightarrow c_a.stg(i) = c_b.stg(i)$
- Given that $\forall j < i, c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$, we get $s \sqsubset_{\mathcal{S}} c_a.stg(j) \Rightarrow s \sqsubset_{\mathcal{S}} c_b.stg(j)$.

(d) $\neg c_a.pending(i, op) \Rightarrow \neg c_b.pending(i, op)$
From $c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$, we get:
- $\neg\exists j < i\,.\,(opc(j) = op \wedge c_a.stg(j) \sqsubset_{\mathcal{S}} post) \Rightarrow \neg\exists j < i\,.\,(opc(j) = op \wedge c_b.stg(j) \sqsubset_{\mathcal{S}} post)$
- if $\exists j < i\,.\,(opc(j) = op \wedge c_a.stg(j) \sqsubset_{\mathcal{S}} post)$, then $\neg c_a.pending(i, op) \Rightarrow (lstg(op), 0) \sqsubseteq_{\mathcal{P}} c_a(j) \Rightarrow (lstg(op), 0) \sqsubseteq_{\mathcal{P}} c_b(j)$

(e) if $c_a.isnext(i, c_a.stg(i)), \forall s\,.\,c_a.nstg(i) \sqsubseteq_{\mathcal{S}} s$,
$\#\{j < i | c_a.nstg(i) \sqsubseteq_{\mathcal{S}} c_a.stg(j) \sqsubseteq s\}$
$\geq \#\{j < i | c_b.nstg(i) \sqsubseteq_{\mathcal{S}} c_b.stg(j) \sqsubseteq s\}$
- From $c_a.cnt(i) = 0$ and $c_a.isnext(i, c_a.stg(i))$, we get: $\forall j < i, c_a.nstg(i) \sqsubseteq_{\mathcal{S}} c_a.stg(j)$.
- Since $\forall j < i, c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$, the number of instructions j between stages $c_a.nstg(i)$ and s must be lower in c_b than in c_a.

(f) if $c_a.isnext(i, \text{IF}), c_a.slot(\text{IF}) \Rightarrow c_b.slot(\text{IF})$
This follows from (e), from $c_a.free(\text{ID}) \Rightarrow c_a.free(\text{ID})$ (that will be shown below) and from the fact that $\forall j < i$, $c_a(i) \sqsubseteq_{\mathcal{P}} c_b(j)$.

(g) if $c_a.isnext(i, \text{IS}), c_a.slot(\text{IS}) \Rightarrow c_b.slot(\text{IS})$
- From statement (e), we get that $\#\{j|\text{IS} \sqsubseteq_{\mathcal{S}} c_a.stg(j) \sqsubseteq_{\mathcal{S}} \text{CO}\} < iq\_size \Rightarrow \#\{j|\text{IS} \sqsubseteq_{\mathcal{S}} c_b.stg(j) \sqsubseteq_{\mathcal{S}} \text{CO}\} < iq\_size$.
- Otherwise, $c_a.slot(\text{IS})$ implies that the $iqueue$ is full, that is $\#\{j|\text{IS} \sqsubseteq_{\mathcal{S}} c_a.stg(j) \sqsubseteq_{\mathcal{S}} \text{CO}\} < iq\_size$. Then $\#\{j|\text{IS} \sqsubseteq_{\mathcal{S}} c_b.stg(j) \sqsubseteq_{\mathcal{S}} \text{CO}$ is either equal to or lower than iq_size. If it is equal, that means that IS contains the same instructions in c_a as in c_b. If $\exists j' < i\,.\,c_a.isnext(j', \text{CO})$, we must have $c_b.isnext(j', \text{CO})$ because $c_a(j') \sqsubseteq_{\mathcal{P}} c_b(j')$. Otherwise, we have $\#\{j|\text{IS} \sqsubseteq_{\mathcal{S}} c_b.stg(j) \sqsubseteq_{\mathcal{S}} \text{CO} < iq\_size$.
- Based on these observations and on $c_a.free(\text{ST}) \Rightarrow c_b.free(\text{ST})$ (shown below), we prove the statement.

(h) if $c_a.isnext(i, \text{ID}), c_a.slot(\text{LSU}) \Rightarrow c_b.slot(\text{LSU})$
- From statement (e), we get that $\#\{j|c_a.stg(j) = \text{LSU}\} < mq\_size \Rightarrow \#\{j|c_b.stg(j) = \text{LSU}\} < mq\_size$
- Otherwise, $c_a.slot(LSU)$ implies that the $mqueue$ is full, that is $\#\{j|c_a.stg(j) = \text{LSU}\} = mq\_size$. Then $\#\{j|c_b.stg(j) = \text{LSU}\}$ is either equal to or lower than mq_size. If it is equal, that means that stages LSU contains the same instructions in c_b as in c_a. If $\exists j' < i\,.\,c_a.isnext(j', \text{LSU})$, we must have $c_b.isnext(j', \text{LSU})$ because $c_a(j') \sqsubseteq_{\mathcal{P}} c_b(j')$. Otherwise, we have $\#\{j|c_b.stg(j) = \text{LSU}\} < mq\_size$.
- Based on these observations and on $c_a.free(\text{LU}) \Rightarrow c_b.free(\text{LU})$ and $c_a.free(\text{SU}) \Rightarrow c_b.free(\text{SU})$ (shown below), we prove the statement.

(i) $c_a.slot(\text{SU}) \Rightarrow c_b.slot(\text{SU})$
- From statement (e), we get that $\#\{j|opc(j) = store \wedge \text{LSU} \sqsubseteq_{\mathcal{S}} c_a.stg(j) \sqsubseteq_{\mathcal{S}} post\} < sq\_size \Rightarrow \#\{j|opc(j) = store \wedge \text{LSU} \sqsubseteq_{\mathcal{S}} c_b.stg(j) \sqsubseteq_{\mathcal{S}} post\} < sq\_size$.
- Otherwise, $c_a.slot(\text{SU})$ implies that the $iqueue$ is full, that is $\#\{j|opc(j) = store \wedge \text{LSU} \sqsubseteq_{\mathcal{S}} c_a.stg(j) \sqsubseteq_{\mathcal{S}} post\} <= sqsize$. Then $\#\{j|opc(j) = store \wedge \text{LSU} \sqsubseteq_{\mathcal{S}} c_b.stg(j) \sqsubseteq_{\mathcal{S}} post\}$ is either equal to or lower than sq_size. If it is equal, that means that stages LSU, SU, CO and ST contain together the same store instructions in c_a as in c_b. If $\exists j' < i\,.\,c_a(j') = (\text{ST}, 0)$, this instruction has either the same state in c_b or it has left the queue. In both cases, this ensures that a slot will be available in the $squeue$ next cycle.

(j) $(j < i.dep(i, j) \wedge c_a.stg(j) \sqsupseteq_{\mathcal{S}} \text{CO}) \Rightarrow c_b.stg(j) \sqsupseteq_{\mathcal{S}} \text{CO}$
This follows from the fact that $\forall j < i.c_a(j) \sqsubseteq_{\mathcal{P}} c_b(j)$

We get $c_a.ready(i) \Rightarrow c_b.ready(i)$ from statements (a), (c), (d) and (j).
If $s \in \{\text{PC}, \text{ID}, \text{IS}, \text{DIV}, \text{LU}, \text{SU}, \text{ST}\} \wedge \neg\exists j.c_a.stg(j) = s$ then, since $\forall j < i, c_a.stg(j) \sqsubseteq_{\mathcal{P}} c_b.stg(j)$, we get $\neg\exists j\,.\,c_b.stg(j) = s$ and thus $c_b.free(s)$. Otherwise, if $\exists j\,.\,c_a.stg(j) = s \wedge c_a.ready(j) \wedge c_a.free(c_a.nstg(j)))$, this instruction is either in the same configuration in c_b, or it has more progress in c_b than c_a and thus $\neg\exists i\,.\,c_b.stg(i) = s$, which leads to $c_b.free(s)$.
From this observation and from statements (f), (g), (h) and (i), we get $c_a.free(c_a.nstg(i)) \Rightarrow c_b.free(c_b.nstg(i))$.

7.2 Proof of Theorem 5 for MINOTAuR

Let c be the state that splits upon the cache uncertainty of instruction i, leading to the hit-case successor state c_b and miss-case successor c_w. Let $mlat$ be the latency of an access to the memory after a cache miss.

- We first consider a data cache miss. As long as $store$ and $atomic$ instructions are pending, i is stalled in the LSU stage in the pipeline states following c_w. Let T denote the number of cycles until pending $store/atomic$ instructions finish their execution. The number of these pending instructions is upper bounded by the size of the $squeue$. Then $T \leq sq\_size * mlat$. After T cycles, the $load$ that was stalled can advance to the LU stage. After $mlat$ additional cycles, it reaches progress $c'_w(i) = (\text{LU}, 0)$ which is equal to $c_b(i)$. We must now show that $c_b \sqsubseteq c'_w$. It follows from:
  - instructions $j < i$ are not affected by the uncertainty on instruction i and thus progressed at least as much in c'_w than in c_b.
  - by the definition of $ready$ and $free$, instructions $k > i$ that progressed in c_b could also progress at least during the cycle transition leading to c'_w.

  The claims follows from $c_b \sqsubseteq c'_w$ and Theorem 2. The maximum penalty of a cache miss is given by $p = (sq\_size + 1) * mlat$.
- We now consider an instruction cache miss. Instruction i is stalled in the PC stage as long as a memory instruction is pending. There can be as many as $iq\_size + 2 + mq\_size + sq\_size$ such pending instructions. After $T = (iq\_size + 2 + mq\_size + sq\_size) * mlat$ cycles at most, the instruction cache miss can be served with an additional $mlat$-cycle latency. The remainder of the proof is analogous to the data cache case.

REFERENCES

[1] 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.

[2] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 2014.

[3] 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.

[4] 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*, pages 13–18, 2006.

[5] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

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

[7] A. Gruin, T. Carle, H. Cassé, and C. Rochange. Gitlab repository for MINOTAuR sources and experiments. https://gitlab.irit.fr/minotaur/MINOTAuR.

[8] 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)*, pages 393–404, 2021.

[9] 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.

[10] 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.

[11] S. Hahn and J. Reineke. Design and analysis of SIC: A provably timing-predictable pipelined processor core. *Real Time Systems*, 2020.

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

[13] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis: definition and challenges. *ACM SIGBED Review*, 12(1):28–36, 2015.

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

[15] 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.

[16] 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.

[17] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium*, 1999.

[18] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1), 2016.

[19] T. Mitra. Time-predictable computing by design: Looking back, looking forward. In *Annual Design Automation Conference*, 2019.

[20] T. Mitra, J. Teich, and L. Thiele. Time-critical systems design: A survey. *IEEE Design and Test*, 35(2), 2018.

[21] 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)*, 2006.

[22] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. T-CREST: time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 2015.

[23] 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.

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

[25] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Fifth International Conference on Quality Software*, 2005.

[26] 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)*, pages 428–441. IEEE, 2018.

[27] 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. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

Alban Gruin is a PhD student at IRIT-Université Toulouse 3. His research focus is on computer engineering, and particularly on the design of timing predictable computer architectures.

Thomas Carle is a lecturer at IRIT-Université Toulouse 3. His research focus is on the timing predictability of embedded real-time systems, especially in parallel processors (multi/many-cores, GPUs), through the use of compilation, static analysis and static scheduling techniques.

Christine Rochange is a professor at IRIT-Université Toulouse 3. Her research interests are on static WCET analysis, more specifically on the modeling of processor and GPU architecture.

Hugues Cassé is an associate-professor at IRIT-Université Toulouse 3 where he teaches machine architecture, compilation and embedded systems. His main research interest concerns static program analysis and worst-case execution time calculation.

Pascal Sainrat is a professor at IRIT-Université Toulouse 3 and the university vice-president for information technology. His research interest is on computer and more specifically processor architecture for critical systems.