Speculative Execution and Timing Predictability in an Open Source RISC-V Core

Alban Gruin, Thomas Carle, Hugues Cassé, Christine Rochange

To cite this version:

Alban Gruin, Thomas Carle, Hugues Cassé, Christine Rochange. Speculative Execution and Timing Predictability in an Open Source RISC-V Core. IEEE Real-Time Systems Symposium (RTSS 2021), Dec 2021, Dortmund, Germany. pp.393-404, 10.1109/RTSS52674.2021.00043. hal-03477573

HAL Id: hal-03477573

https://hal-univ-tlse3.archives-ouvertes.fr/hal-03477573

Submitted on 13 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Speculative Execution and Timing Predictability in an Open Source RISC-V Core

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

***Abstract*—We present MINOTAuR, a timing predictable open source RISC-V core based on the Ariane core [28]. We first modify Ariane in order to make it timing predictable following the approach used to design the SIC processor [12]. We prove that the instruction parallelism in the Ariane core does not prevent from enforcing timing predictability. We further relax restrictions by enabling a limited amount of speculative execution and we are still able to formally prove that the core is timing predictable. Experimental results show that the performance is reduced by only 10% on average compared to the original Ariane core.**

***Index Terms*—timing predictability, processor architecture**

I. Introduction

Modern real-time systems require an increasing amount of processing power, leading to the adoption of multi-core processors as well as single-core processors featuring acceleration mechanisms which combine complex pipelines, out-of-order execution, cache memories and branch speculation. Since the worst-case execution time (WCET) analysis [1], [4], [27] of a real-time system requires a precise model of the timing behavior of the hardware, this complexity makes it more difficult to compute safe and precise WCET bounds.

The traditional approach to WCET computation involves successive analyses performed on the control flow graph of the program. In the presence of conditional branches or loops, these analyses can identify uncertain outcomes, even for simple programs: for example, fetching an instruction can lead to a miss or a hit in the instruction cache depending on the execution path that led to this instruction. Since the objective of WCET analysis is to obtain a worst-case bound, one may think that, in such situations, the problem can be solved by always considering the worst local case (e.g. cache miss). However, it has been shown [22] that due to a phenomenon called *timing anomalies*, a local best case can lead to the global worst case (i.e. a longer execution time). As a result, considering only the local worst cases is not a safe approach.

Timing anomalies make the WCET analysis more difficult for single-core architectures: it must take into account all the possible combinations of uncertain outcomes, leading to a combinatorial increase of the analysis time. Moreover, in multi-core architectures, the possible occurrence of timing anomalies further increases the algorithmic complexity of interference analysis: multiple cores that share a resource may lengthen the execution time of one another due to the sequential access to the shared resource. Such delays may be amplified inside a pipeline by timing anomalies. As a result, the only safe way to temporally analyze a real-time system implemented on a multi-core processor that may exhibit timing anomalies is to consider a cycle-accurate representation of all the possible executions of the programs on the various cores, which is intractable in practice.

To overcome these difficulties, the strictly in order (SIC) core [11] approach proposes (i) structural modifications that suppress the possibility 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 lead 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 reported to suffer a 7% drop of performance compared to 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 [11]: the open source RISC-V Ariane [28] 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.

Our main contributions are the following:

- we provide a formal model of the MINOTAuR$_\beta$ core obtained by applying the same restrictions as SIC on the Ariane core. We prove its timing predictability and we evaluate its performance on a cycle-accurate simulator: the loss is 41.2% compared to Ariane.
- we provide a formal model of the MINOTAuR core derived from MINOTAuR$_\beta$ by partially relaxing the re-

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

strictions on speculative execution. We prove that it is also timing predictable and that the performance gap w.r.t. Ariane is reduced to only 10% on average.

The remainder of the paper is organized as follows. Section II presents the state of the art regarding timing predictable processors and introduces in more details the SIC approach. Section III describes the internal organization of the Ariane core and presents our experimental methodology and first results. We introduce the MINOTAuR$_\beta$ core in Section IV and the MINOTAuR core in Section V. Finally, Section VI concludes the paper and presents the future work perspectives.

II. Related work

A. Timing predictability

A processor is said to be *timing predictable* when it is free of timing anomalies and timing compositional [11]. 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) [18]. This makes the timing analysis more complex since all the possible situations have to be considered. Several authors have investigated this phenomenon, proposing various definitions and means to detect whether a processor is prone to such timing anomalies [2], [6], [8], [22], [26]. It turns out that most of the off-the-shelves cores, even the simplest ones, may suffer from timing anomalies, which motivates the design of timing-anomalies-free processors (see Section II-B).

Timing compositionality is a property that simplifies the timing analysis of a multi-core system [14]. It allows combining the analysis results for individual components instead of performing a very complex fully-integrated system analysis. An approach to sound and precise compositional timing analysis for multicore systems is proposed in [10].

B. Timing-predictable processor architectures

Several approaches have been considered to enforce timing predictability in hardware platforms [3], [20], [21].

The Kalray MPPA-256 processor [5] has been designed with timing predictability in mind. In addition to its VLIW architecture (initially motivated by power considerations), architectural choices 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 [16] is an implementation of a precision-timed (PRET) machine [17]. It features a repeatable thread-interleaved pipeline that exploits fine-grained thread-level parallelism. 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 [24] 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 [23]. Although it has been designed to be timing predictable, this has not been formally proven to the best of our knowledge.

In [11], [12], Hahn and Reineke introduce SIC, a strictly in-order core, and show that it is free of timing anomalies and timing compositional. We summarize their formal framework used to prove these two properties in Section II-C. SIC is a simple 5-stage in-order pipelined processor in which the fetch of instructions is gated in order to guarantee that an instruction can never be delayed by a younger instruction.

C. A formal framework to prove timing predictability

In [13] a framework to express the concrete semantics of a processor pipeline is proposed. 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 $\mathcal{C} \subseteq \mathcal{I} \rightarrow \mathcal{P}$, 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 $cycle : \mathcal{C} \rightarrow \mathcal{C}$ that relates a pipeline state to its successor.

In [11], 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 [11].

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

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 [11] demonstrate that the SIC processor is free of timing anomalies w.r.t. uncertain cache behaviour, and timing-compositional w.r.t. 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 engenders 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 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, 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 w.r.t. 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 [11].

Theorem 5. Compositionality 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. 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 [11] is specific to the SIC processor.

III. The Ariane RISC-V core

A. The Ariane architecture

The Ariane core [28] is a RISC-V 6-stage in-order processor. The address of the next instruction to be fetched is computed in the first stage (PC). The instruction fetch (IF) stage hosts four branch predictors: 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). 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). The fetched instructions are inserted in an instruction queue which they exit in the instruction decode (ID) stage. This queue has a capacity of 4 instructions.

A scoreboard contains all decoded instructions until they are committed. It can contain up to 8 instructions. The issue stage (IS) inserts instructions in the scoreboard and sends 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 buffer (that contains instructions that access Control/Status Registers). 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 contains a load unit (LU) and a store unit (SU). All memory instructions spend at least one cycle in a queue (which

can hold at most 2 instructions) in 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 them in a store buffer (that has a capacity of 4 instructions). 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 [25]. It does not rename registers, has no MMU, no FPU, and has a single commit port. Its CoreMark score is reported in Table II.

B. Experimental methodology

All the results reported in this paper have been obtained using a SystemVerilog model of the processor, simulated with the Questa Advanced Simulator 10.7g[1].

As benchmarks, we have used the `kernel` and `sequential` sets of programs of the TACLe benchmark suite[2] [7] as well as the CoreMark[3], all compiled with `gcc 10.2.0` and optimization flag `-O2`.

We provide individual results for each benchmark, as well as average results computed over the set of benchmarks: the arithmetic mean and a weighted arithmetic mean where the weight of a benchmark reflects its number of executed instructions.

The source code for all cores and experiments presented in this paper is available at [9].

C. 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) [14]. We refer to this phenomenon as an *inversion*.

[1]eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/

[2]We had to exclude `ammunition`, `lms`, `mpeg2`, `rijndael_dec`, `susan` which did not compile or execute on the simulator

[3]www.coremark.org

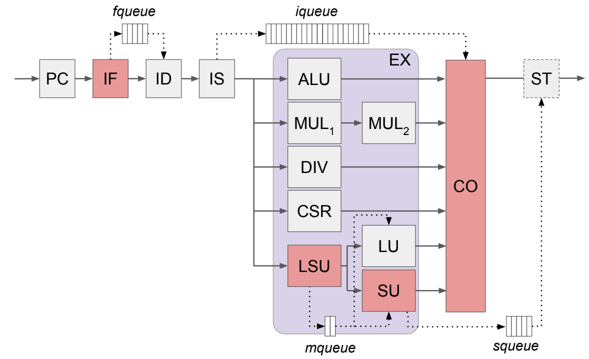

Fig. 1. Model of the Ariane core pipeline

To get an insight into whether the Ariane core might be prone to timing anomalies, we have performed measurements to determine whether we could observe such inversions. 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 III-B. Results are reported in Table I. Over 52 TACLe benchmarks, 26 had inversions during their execution. This reveals that Ariane is potentially subject to timing anomalies and motivates our work to make it timing predictable.

IV. MINOTAuR$_\beta$: A TIME COMPOSITIONAL RISC-V CORE WITH PARTIALLY OUT-OF-ORDER EXECUTION

A. Model of the Ariane core

Before giving a formal description of the MINOTAuR$_\beta$ core, we introduce our model of the Ariane pipeline. This model is depicted in Figure 1. It includes the pipeline stages that we mentioned in Section III-A (PC, IF, ID, IS, CO) and the FUs are grouped in an execution stage, EX. The branch unit is included in the ALU, which it uses to perform comparisons. The multiplier/divider is modelled as two entities: a 2-stage multiplier (MUL$_1$ and MUL$_2$) and a separate divider (DIV). The CSR buffer is also modelled as a separate FU. The memory units are represented as a LSU unit followed by separate LU and SU units.

The Ariane core features several instructions queues that we model in the following way. We consider that an instruction that resides in a queue stays fictitiously 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 modelled 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 fictitiously hosted) are shown in light red in Figure 1.

B. Formal model of MINOTAuR$_\beta$

The MINOTAuR$_\beta$ core is obtained from the baseline Ariane processor by applying the same restrictions as in the SIC paper: the branch predictors are disabled so that any branch will stall the pipeline in the PC stage, and misses in the instruction cache are stalled (and thus they do not perform their bus transactions) as long as there is a memory instruction in the pipeline.

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$_\beta$ core is shown in Figure 2. This model specifies the pipeline structure[4] 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 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 fictitiously 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 required functional unit must be available right after the cycle spent in IS, conflicts on the result bus must be prevented and the instruction is stalled if a *csr* instruction is pending), 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.

In order to save space, Figure 2 highlights the variations corresponding to the MINOTAuR$_\beta$ core in orange, and the ones corresponding to the MINOTAuR core in blue. In particular, in the MINOTAuR$_\beta$ model, function $nstg$ is equal to function $nstg'$. The part corresponding to the modifications made to the baseline Ariane core is the orange portion in the $c.ready(i)$ function: (i) the PC stage is stalled whenever a branch instruction is already in the pipeline (and has not reached the end of the ALU stage, in which the target address and/or branch condition are resolved), and (ii) the instructions do not enter the IF stage if they do not result in a hit in the instruction cache and if a previous memory instruction is already in the pipeline.

C. Anomaly freedom and compositionality proofs

Theorem 6. Timing predictability of MINOTAuR$_\beta$. *The MINOTAuR$_\beta$ core is free of timing anomalies and is timing compositional.*

Proof. We prove in the Appendix that Property 1 holds for MINOTAuR$_\beta$. As a consequence, Lemmas 1 and 2, and Theorems 1 to 4 also hold. It remains to prove Theorem 5 for MINOTAuR$_\beta$, which is also done in the Appendix. ☐

D. Performance evaluation

We followed the methodology described in Section III-B to evaluate the performance of MINOTAuR$_\beta$.

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

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

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

$$cycle(c)(i) := \begin{cases} (c.nstg(i), c.nlat(i)) & : c.ready(i) \wedge c.free(c.nstg(i)) \\ (c.stg(i), c.ncnt(i)) & : otherwise \end{cases} \qquad c.isnext(s, i) := c.stg(i) = s \wedge \forall j < i \,.\, c.stg(j) \sqsupset_{\mathcal{S}} s$$

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

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

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

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

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

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

$$\begin{aligned} c.ready(i) := \; & (c.stg(i) \neq pre \wedge \neg c.pending(i, branch) \wedge pwrong(i)) \\ & \vee (c.cnt(i) = 0 \wedge c.isnext(c.stg(i), i)) \\ & \quad \wedge (c.stg(i) = \text{PC} \Rightarrow \neg c.pending(i, branch) \wedge (ichit(i) \\ & \qquad \vee (\neg c.pending(i, load) \wedge \neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \\ & \quad \wedge (c.stg(i) = \text{PC} \Rightarrow (ichit(i) \\ & \qquad \vee (\neg c.pending(i, branch) \wedge \neg c.pending(i, load) \wedge \neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \\ & \quad \wedge (c.stg(i) = \text{ID} \Rightarrow \\ & \qquad ((opc(i) \in \{load, store, atomic\} \Rightarrow c.slot2(\text{LSU})) \\ & \qquad \wedge (opc(i) \notin \{load, store, atomic\} \Rightarrow \neg c.pending(i, div) \wedge \neg c.pending(i, csr)) \\ & \qquad \wedge (opc(i) = mul \Rightarrow \neg c.pending(i, mul)))) \\ & \quad \wedge (c.stg(i) = \text{LSU} \Rightarrow (opc(i) \in \{store, atomic\} \wedge \neg c.pending(i, atomic)) \\ & \qquad\qquad \vee (opc(i) = load \wedge (\neg c.pending(i, store) \wedge \neg c.pending(i, atomic)))) \end{aligned}$$

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

$$\begin{aligned} c.slot(\text{IF}) := \; & (\#\{j | c.stg(j) = \text{IF}\} < fq_size) \vee c.free(\text{ID}) \\ c.slot(\text{IS}) := \; & \#\{j | \text{IS} \sqsubseteq_{\mathcal{S}} c.stg(j) \sqsubseteq_{\mathcal{S}} \text{CO}\} < iq_size \vee (\exists j' \,.\, c.isnext(\text{CO}, j') \wedge c.ready(j') \wedge (opc(j') \in \{store, atomic\} \Rightarrow c.free(\text{ST}))) \\ c.slot(\text{SU}) := \; & \#\{j | opc(j) = store \wedge \text{LSU} \sqsubset_{\mathcal{S}} c.stg(j) \sqsubset_{\mathcal{S}} post\} < sq_size \vee \exists j' \,.\, c(j') = (\text{ST}, 0)) \\ c.slot(\text{LSU}) := \; & \#\{j | c.stg(j) = \text{LSU}\} < mq_size \\ & \vee (\exists j' \,.\, c.isnext(\text{LSU}, j') \wedge ((opc(j') = load \wedge c.free(\text{LU})) \vee (opc(j') \in \{store, atomic\} \wedge c.free(\text{SU})))) \\ c.slot2(\text{LSU}) := \; & \#\{j | opc(j) \in \{load, store, atomic\} \wedge \text{IS} \sqsubseteq_{\mathcal{S}} c.stg(j) \sqsubseteq_{\mathcal{S}} \text{LSU}\} < mq_size \\ & \vee (\exists j' \,.\, c.isnext(\text{LSU}, j') \wedge ((opc(j') = load \wedge c.free(\text{LU})) \vee (opc(j') \in \{store, atomic\} \wedge c.free(\text{SU})))) \end{aligned}$$

Fig. 2. Model of the core. In orange, the MINOTAuR$_\beta$ version. In blue, the MINOTAuR version.

As expected, we did no longer observe any inversion. Compared to the baseline Ariane core, the increase on the execution time ranges from 10.92% to 125.21%, and reaches 41.2% on average (47.5% for the weighted mean). These results are significantly higher than the 6-7% loss reported in [11]. 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 [15] 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.

In order to estimate the impact of these mechanisms, we designed a restrained version of Ariane in which we reduced the size of the *iqueue* to a single slot and disabled the dynamic branch predictor. The goal was to make it as close as possible to the simple pipeline used to build SIC. We refer to this version as *seqAriane*. Table I shows that seqAriane runs

Benchmark	Ariane Inversions	Ariane Cycles	seqAriane Overhead	sicAriane Overhead	MINOTAuR$_\beta$ Overhead	MINOTAuR Overhead
binarysearch	2	2,206	38.08%	67.45%	21.53%	13.83%
bitcount	0	19,924	138.51%	184.80%	49.44%	27.44%
bitonic	0	10,513	150.62%	189.58%	33.21%	7.40%
bsort	0	60,817	210.62%	286.84%	125.21%	1.39%
complex_updates	0	21,153	165.05%	190.24%	39.71%	10.17%
cosf	0	325,912	178.51%	203.03%	41.16%	4.68%
countnegative	0	19,268	69.19%	101.87%	36.54%	11.93%
cubic	3	12,907,490	168.08%	190.13%	38.61%	5.93%
deg2rad	0	172,914	160.36%	183.77%	39.67%	5.86%
fac	0	1,247	29.35%	53.97%	24.38%	20.13%
fft	1	2,178,836	167.34%	191.26%	47.29%	11.47%
filterbank	1	50,353,639	172.27%	196.56%	36.12%	5.13%
fir2dim	0	36,012	158.18%	182.58%	36.07%	6.48%
iir	2	6,326	116.20%	146.38%	36.29%	15.40%
insertsort	1	2,101	79.82%	127.27%	41.41%	26.23%
isqrt	0	524,958	141.57%	169.22%	50.71%	0.67%
jfdctint	0	4,884	80.04%	106.94%	25.43%	25.31%
ludcmp	1	62,314	161.09%	194.98%	34.94%	6.23%
matrix1	0	12,748	181.66%	226.58%	54.63%	4.03%
md5	0	7,452,238	166.99%	196.97%	30.53%	4.42%
minver	4	22,826	124.38%	147.55%	28.28%	11.19%
pm	16	120,964,347	192.28%	216.68%	51.10%	4.05%
prime	0	1,478	26.32%	53.59%	31.39%	24.76%
quicksort	2	4,427,652	159.97%	202.32%	48.22%	8.94%
rad2deg	0	171,814	164.44%	189.48%	42.87%	6.45%
recursion	0	3,081	84.45%	131.55%	40.93%	20.35%
sha	0	928,238	189.52%	245.28%	57.01%	3.91%
st	1	1,924,247	183.54%	206.41%	48.90%	8.63%
adpcm_dec	3	142,166	79.76%	91.24%	29.94%	1.11%
adpcm_enc	0	146,493	76.62%	87.89%	29.34%	3.00%
anagram	4	1,557,582	168.95%	220.02%	41.75%	7.94%
audiobeam	2	4,000,942	159.77%	182.04%	35.35%	4.98%
cjpeg_transupp	0	1,993,028	170.43%	218.58%	78.70%	5.23%
cjpeg_wrbmp	0	80,845	126.27%	169.87%	30.13%	14.72%
dijkstra	1	35,825,901	143.76%	201.96%	76.49%	9.38%
epic	3	40,198,007	177.47%	200.52%	39.21%	3.65%
fmref	1	7,220,481	168.66%	190.79%	36.28%	3.64%
g723_enc	2	549,627	144.96%	167.91%	54.72%	4.13%
gsm_dec	2	1,319,645	150.11%	167.45%	58.10%	12.80%
gsm_enc	7	4,109,217	154.36%	183.35%	24.49%	10.51%
h264_dec	2	236,863	111.61%	139.64%	37.81%	22.40%
huff_dec	1	113,176	113.22%	143.27%	47.12%	10.62%
huff_enc	3	479,586	138.12%	179.34%	34.41%	10.02%
ndes	1	58,675	127.58%	163.45%	22.85%	7.35%
petrinet	0	1,660	20.84%	40.60%	22.11%	19.58%
rijndael_enc	5	4,652,960	193.56%	220.88%	10.92%	3.73%
statemate	4	37,468	207.54%	288.99%	36.47%	12.86%
Average			138.1%	171.1%	41.2%	10.0%
Weighted mean			176.5%	205.4%	47.5%	5.2%

TABLE I

RESULTS OF THE TACLE BENCHMARK SUITE RAN ON ALL VERSIONS OF THE CORE. THE OVERHEADS ARE ALL COMPUTED W.R.T. ARIANE.

	Ariane	seqAriane	Overhead	sicAriane	Overhead	MINOTAuR$_\beta$	Overhead	MINOTAuR	Overhead
CoreMark score	110.753173	46.541674	57.98%	41.099235	62.89%	71.870817	35.1%	106.670535	3.68%

TABLE II

COREMARK SCORE.

noticeably slower than Ariane: execution times are increased by 138.1% on average on the TACLe benchmarks.

We also designed another version of seqAriane that implements the gating mechanism used to make SIC timing predictable (instruction fetches are stalled as long as a memory instruction or an unresolved branch instruction is pending). We refer to this version as *sicAriane*. As expected, the performance is further degraded compared to Ariane: execution times reported in Table I show an average increase of 171.1% on average on the TACLe benchmarks. Compared to seqAriane, the increase is 14.17%[5] which is still higher than the 6-7% loss reported for SIC. This probably stems from differences in the microarchitecture. We observe that MINOTAuR$_\beta$ performs 92.33% better than sicAriane on average on the TACLe benchmarks. Table II shows that it outperforms sicAriane by

[5]This value has been calculated from the raw numbers of cycles that we could not display due to space limitations but that can be derived from the data given in the table.

74.8% on the CoreMark. This indicates that we were able to transpose the approach proposed in [11] to a more complex processor. However, as mentioned above, the performance of MINOTAuR$_\beta$ is significantly lower than that of Ariane. Our intuition is that disabling speculative execution strongly limits the performance in a core that can efficiently process instructions. We relax this restriction in the next section.

V. MINOTAuR: A speculative time predictable RISC-V core

In order to reduce the performance loss, we designed a new version of the MINOTAuR$_\beta$ core in which the branch prediction mechanisms are active and speculation is enabled to a certain extent. However, we decided to disable the RAS because its behavior could incur timing anomalies[6]. It is left for future work to design a RAS which provably does not incur timing anomalies. The idea is to let the core execute speculatively as long as the execution does not modify the state of the hardware (e.g. cache content) other than the instructions in the pipeline. We refer to the resulting core as MINOTAuR. In the following, we present how these modifications affect the model. Then we prove that MINOTAuR is timing predictable, and finally we show experimentally that the execution time overhead is considerably reduced.

A. Model

The modifications corresponding to MINOTAuR are highlighted in blue in Figure 2.

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. We introduce a new predicate, $pwrong(i)$, that is true whenever instruction i is misspeculated. Using this predicate, we extend the $nstg$ function so that 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. The actual relaxation on the speculation appears 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. To complete this model, the *free* function is extended in the exact same fashion.

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 the next section, we will prove that it is the case in the MINOTAuR core. However, we point out right away that this property is sensitive to the cache write and replacement policy: it works with any cache in which hits do not modify the cache state[7] (e.g. direct-mapped caches or random caches such as the ones implemented in MINOTAuR). This would not be the case with caches implementing ageing mechanisms, such as LRU caches. Designing efficient speculation-insensitive LRU caches is left for future work. Moreover, a speculated store instruction cannot perform its write to memory (in stage ST, i.e. after stage CO) before the corresponding branch instruction is resolved. Thus we do not need to consider the effect of stores in our proofs.

[6]The RAS incurs two difficulties compared to the other speculative mechanisms handled in the paper: first, it is sometimes implemented as a circular buffer which looses information when too many function calls are nested; second, 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.

B. 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:

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

Theorem 7 (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 $c.stg(i) \sqsubset_{\mathcal{S}}$ PC. Then trivially, $cycle(c).stg(i) \sqsubset_{\mathcal{S}}$ IF. Let us now consider $c \in \mathcal{C}$ such that $c.stg(i) =$ 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 $cycle(c).stg(i) =$ PC. We can recursively apply the same argument to prove that for all states c' such that $c'.pending(i, branch)$, $cycle(c').stg(i) \sqsubset_{\mathcal{S}}$ IF. From this we conclude that (1) does not hold.

Now let us assume 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 $c.stg(i) \sqsubseteq_{\mathcal{S}} c.stg(j_{br}) \sqsubseteq_{\mathcal{S}}$ IS. If $c.stg(j_{br}) \sqsubset_{\mathcal{S}}$ IS, then trivially $cycle(c).stg(i) \sqsubset_{\mathcal{S}}$ LSU and (2) does not hold. If $c.stg(j_{br}) =$ IS, then necessarily $c.stg(i) \sqsubset_{\mathcal{S}}$ IS and once again $cycle(c).stg(i) \sqsubset_{\mathcal{S}}$ 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 7 remains valid for any instruction i as long as there exists an unresolved branch instruction which precedes i.

In this proof we showed that no memory access is performed speculatively: it results that (i) no request to the

[7]More precisely: for which the effect of cache hits is transparent to usual cache analysis [19]

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 the rest is unchanged w.r.t. Section IV.

Theorem 8 (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 $c_a.ready(i) \Rightarrow c_b.ready(i)$. The property was proven in all cases but the two blue ones (see the Appendix), so we focus only on these two cases here.

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 Theorems 7 and 8, we conclude that the results of Section IV-C still apply on MINOTAuR, and 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 the relaxation of the constraint on speculation 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 9 (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:*

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

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 7 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 9, 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 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.*

Proof. We use the same notations as for Theorem 9. 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.

□

Finally we can bound p by the worst-case time that a branch can take to be resolved: it is the worst-case number of cycles an instruction can spend between the PC and the ALU stages. We denote $divlat$ the worst-case latency of a division and $mlat$ the duration of an access to the memory after a cache miss. In our core, $divlat > mlat$. The worst-case traversal time is observed when the branch enters stage PC while the instructions residing between stages IF and ID have the worst possible latency in their functional units ($divlat$) and the entrance into the scoreboard is delayed by the worst possible configuration: a division instruction in the DIV functional unit. When the first division instruction exits the DIV unit, the second one is allowed to enter the IS stage, thus one additional cycle is required. The same is true for all the divisions residing in the $fqueue$ and for the branch itself. Since $divlat > mlat$, a miss in the instruction cache would not contribute to the worst-case latency. In the end $p \leq ((2 + fq_size) * (divlat + 1) + 1)$ cycles.

C. Performance evaluation

Experimental results for MINOTAuR are available in Tables I and II. We followed the methodology described in Section III-B.

Again, we did not observe any inversion, which was expected due to the gating mechanism that we have implemented. Due to the fact that we have carefully selected the restrictions that were absolutely required to prove timing predictability and relaxed the other ones, the performance loss compared to the original Ariane core is noticeably low: the increase is 3.68% on the CoreMark, and ranges from 0.67% to 27.44%, with an average of 10.0% on the TACLe benchmarks. By relaxing the limitations on speculative execution, we thus claimed back 30% of the performance on average (compared to MINOTAuR$_\beta$), while keeping the core provably timing predictable.

The cost in performance may look higher than for the SIC core at first sight, but as mentioned before, our baseline core is much faster than the one considered for SIC. From the numbers given in Table I, we can derive that MINOTAuR is more than twice faster on average than sicAriane, which is supposedly comparable to SIC : the execution time on sicAriane is 148.96% longer than on MINOTAuR.

VI. Conclusion

In this paper we presented MINOTAuR, a timing predictable core based on the open source Ariane RISC-V core. We first applied the SIC philosophy [11] on Ariane and found experimentally that the resulting performance degradation was substantial (41.2% on average on the TACLe benchmarks). We thus relaxed part of the limitations on branch speculation and showed that we could still formally prove timing predictability. The cost for this predictability, i.e. the execution time overhead of MINOTAuR compared to the baseline Ariane core, is only 10% on average. This shows that timing predictability is compatible with acceleration mechanisms such as dynamic branch prediction and parallel functional units, 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 the future we plan on extending MINOTAuR with new mechanisms that will not impair its provable timing predictability: speculation-insensitive LRU caches whose state is not altered by wrong branch predictions, a predictable return address stack, and a scheme that will allow multiple functional units to execute instructions in parallel.

VII. Appendix

A. Proof of Property 1 for MINOTAuR$_\beta$

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) and from $c_a.free(\text{ID}) \Rightarrow c_a.free(\text{ID})$ (that will be shown below).

(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[8], 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) if $c_a.isnext(i, \text{ID})$, $c_a.slot2(\text{LSU}) \Rightarrow c_b.slot2(\text{LSU})$

- From statement (e), we get that $\#\{j | opc(j) \in \{load, store, atomic\} \wedge c_a.stg(j) \in \{\text{IS}, \text{LSU}\}\} < mq_size \Rightarrow \#\{j | opc(j) \in \{load, store, atomic\} \wedge c_b.stg(j) \in \{\text{IS}, \text{LSU}\}\} < mq_size$
- Otherwise, $c_a.slot2(\text{LSU})$ implies that the $mqueue$ is full, that is $\#\{j | opc(j) \in \{load, store, atomic\} \wedge c_a.stg(j) \in \{\text{IS}, \text{LSU}\}\} = mq_size$. Then $\#\{j | opc(j) \in \{load, store, atomic\} \wedge c_b.stg(j) \in \{\text{IS}, \text{LSU}\}\}$ is either equal to or lower than mq_size. If it is equal, that means that stages IS and LSU contain together 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 $\#\{j | opc(j) \in \{load, store, atomic\} \wedge c_b.stg(j) \in \{\text{IS}, \text{LSU}\}\} < mq_size$.
- 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.

(j) $c_a.slot(\text{SU}) \Rightarrow c_b.slot(\text{SU})$

- 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.

We get $c_a.ready(i) \Rightarrow c_b.ready(i)$ from statements (a), (c), (d) and (i).

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 (j), we get $c_a.free(c_a.nstg(i)) \Rightarrow c_b.free(c_b.nstg(i))$.

B. *Proof of Theorem 5 for MINOTAuR$_\beta$*

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 $mlat$

[8] We have voluntarily omitted this condition in the model for the sake of readability

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 $free$, it follows that 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] absInt. absInt aiT. https://www.absint.com/ait/index.htm.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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

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

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

[15] J. Hennessy and D. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th edition, 2011.

[16] 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.

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] ThalesGroup. Cva6-softcore-contest. https://github.com/thalesgroup/cva6-softcore-contest.

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

[27] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008.

[28] 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.