Enabling timing predictability in the presence of store buffers

Alban Gruin, Thomas Carle, Christine Rochange, Pascal Sainrat

Enabling timing predictability in the presence of store buffers

Alban Gruin
alban.gruin@irit.fr
IRIT – Univ. Toulouse 3 – CNRS
Toulouse, France

Thomas Carle
thomas.carle@irit.fr
IRIT – Univ. Toulouse 3 – CNRS
Toulouse, France

Christine Rochange
christine.rochange@irit.fr
IRIT – Univ. Toulouse 3 – CNRS
Toulouse, France

Pascal Sainrat
pascal.sainrat@irit.fr
IRIT – Univ. Toulouse 3 – CNRS
Toulouse, France

ABSTRACT

We study the effect of store buffers on the timing predictability of processor pipelines. We show that the concurrency between the load unit and the store buffer to access the memory bus is an obstacle to timing predictability, even in simple scalar in-order pipeline designs. We then propose a gating mechanism that removes these shortcomings. We model subsets of pipelines that implement our mechanism using an established logic framework. Using the Coq proof assistant, we prove that it enables timing predictability. Finally, we show that the performance cost of our mechanism measured on an FPGA-synthesized RISC-V core is around 2% only on average, with a minimal increase in resource usage.

CCS CONCEPTS

• **Hardware → Theorem proving and SAT solving**; **Safety critical systems**.

KEYWORDS

computer architecture, timing predictability, timing anomalies, Coq

ACM Reference Format:
Alban Gruin, Thomas Carle, Christine Rochange, and Pascal Sainrat. 2023. Enabling timing predictability in the presence of store buffers. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 7–8, 2023, Dortmund, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3575757.3593653

1 INTRODUCTION

The growing complexity of real-time embedded software as well as the search for increased performance lead to the adoption of advanced processors that implement complex hardware acceleration mechanisms such as cache memories, speculative execution and store buffers. At the same time, reliable worst-case execution time (WCET) upper bounds are required to build safe real-time systems. In the past years, it has been shown that even the simplest processor core designs can be prone to *timing anomalies* [18]. A timing anomaly occurs when a shorter latency for one instruction in a sequence (e.g. a cache hit instead of a cache miss for a load instruction) counter-intuitively makes the execution time of the sequence longer, or when a longer latency for one instruction leads to an even longer increase for the total execution time of the sequence. Timing anomalies make usual compositional analyses unsound for multi-core processors [13].

Timing anomalies appear when components (e.g. caches, buffers) concurrently access a shared resource. A series of works have investigated the design of provably timing-anomaly-free cores [6, 8, 12]. The proposed solutions control concurrency by design, through gating mechanisms that ensure that one given component always has priority over another one when accessing a shared resource. The main challenge is to guarantee the absence of timing anomalies while inducing a minimal decrease in performance. In these papers, the main focus has been put on the concurrency between the instruction and the data cache for accesses to the memory bus. Here, we focus on store buffers and show that they jeopardize the timing predictability of cores, in very simple pipeline designs as well as in more complex ones. We propose a new gating mechanism and use the Coq proof assistant [19] to prove that it prevents timing anomalies related to memory accesses. We implemented this mechanism in a RISC-V core that we synthesized on a FPGA, and we measured its impact on the performance of the core.

2 STORE BUFFERS

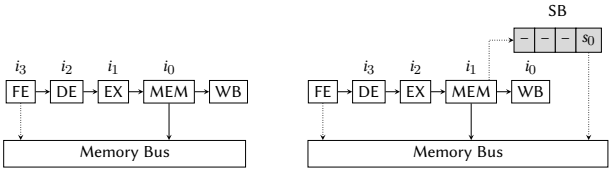
In a processor pipeline, a store buffer [1] allows store instructions to leave the memory stage and be committed while the write request has not been sent to the memory yet. This way, the write latency is hidden.

2.1 In-order pipelines

In a simple scalar in-order pipeline such as the textbook 5-stage pipeline considered in [14], all memory accesses (loads and stores) are processed in the same memory stage. Without a store buffer, a store to a memory block that does not reside in the data cache must stay in the memory stage for at least the latency of the access to the memory, stalling all subsequent instructions. This is illustrated in Figure 1(a): instruction i_0 is a store that accesses the memory bus and stalls the pipeline until the write to the memory is completed. Although this stall is usually necessary for loads (since the loaded data is likely to be used by the following instructions), stores can often be delayed safely.

This work was partially supported by the ANR LabEx CIMI (grant ANR-11-LABX-0040) within the French State Programme "Investissements d'Avenir.".
This work was supported by a grant overseen by the French National Research Agency (ANR) as part of the ProTiPP (ANR-22-CE25-0004) project.

RTNS 2023, June 7–8, 2023, Dortmund, Germany



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

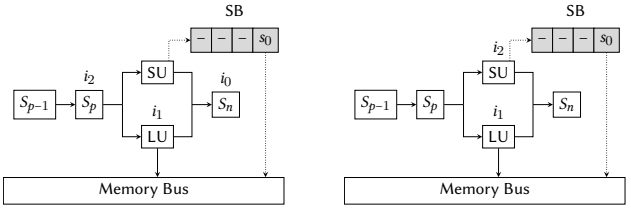
Figure 1: In-order pipeline

A store buffer keeps track of the stores that have been executed in the pipeline but have not yet been sent to the memory. As a consequence, a store instruction to an uncached block does not wait in the memory stage until the block has been loaded or the data has been written to the memory: the address and data to be written are pushed to the store buffer, and the instruction goes to the next stage in the next clock cycle, allowing the flow of instructions to progress in the pipeline. The store buffer then sends the pending writes to memory whenever it gets access to the memory bus. This is illustrated in Figure 1(b): store instruction i_0 is not stalled in the MEM stage. Instead the write (s_0) is enqueued in the store buffer, and i_0 progresses to the WB stage. In the example, instruction i_1 enters the MEM stage and performs a load, thus requesting the memory bus. When the load is completed, i_1 advances to the WB stage and i_2 enters the MEM stage. If i_2 does not request the bus, the store buffer is able to send s_0 to the memory.

2.2 Out-of-order loads and stores

In more complex architectures, load and store instructions to different memory blocks may be executed out of order by using separate instruction queues for loads and stores, or a reorder buffer in the load/store unit (LSU). In this setup, a store buffer allows store instructions to advance in the pipeline even though an older load instruction may be using the memory bus. In order to avoid breaking memory dependencies, addresses of loads are checked against those of the stores in the store buffer. Figure 2 depicts an example of such a design, in which we represent separately the load unit (LU) and the store unit (SU). In order to remain as general as possible, we only depict the portion of the pipeline related to memory accesses. This portion may be inserted in a much longer pipeline and/or in parallel with other functional units. In Figure 2(a), instruction i_0 just advanced to stage S_n after enqueuing its store s_0 in the store buffer in the last cycle. Instruction i_1 (a load) just entered the LU stage and started using the bus, while instruction i_2 (another store) just entered stage S_p. The state of the pipeline in the next cycle is depicted in Figure 2(b): instruction i_0 advances past the S_n stage, while i_1 is still accessing the memory. i_2 is allowed to progress to the SU stage. In the following cycle, i_2 will enqueue its store in the store buffer. If i_1 still has not finished its access to the memory, i_2 will advance to stage S_n.

A consequence of this design is the possibility for a load to be delayed or not before accessing the bus, depending on the state of the store buffer. As we will see in Section 4, this feature is a problem for timing predictability. Additionally, depending on the design, a store instruction may be blocked in the LSU because the



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

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

Figure 2: Out-of-order memory accesses

store buffer is full while a subsequent load may be allowed to use the memory bus, thus delaying the advance of the store instruction. This can also be problematic for timing predictability.

3 GENERAL MODEL DESCRIPTION

In this section, we introduce the formal elements that we use to model the pipelines and reason on their timing predictability properties. Most of these elements are taken from or inspired by [11].

We consider a sequence of instructions $\mathcal{I} = i_0, i_1, ..., i_n$ processed through a pipeline.

Definition 1. ***(Instruction kinds)*** *Each instruction $i \in \mathcal{I}$ has a kind: $i.kind \in \{load, store, other\}$.*

Definition 2. ***(Pipeline)*** *A pipeline is a set of stages $\mathcal{S}$, partially ordered by $\sqsubset_{\mathcal{S}}$.*

Instructions flow through the pipeline stages in an order compatible with $\sqsubset_{\mathcal{S}}$. Note that depending on the pipeline and on their kind, instructions may not flow through all the pipeline stages. The progress of an instruction is a record of the stage it currently resides in and of the number of remaining cycles to process it in this stage.

Definition 3. ***(Progress)*** *The progress of an instruction belongs to $\mathcal{P} := \mathcal{S} \times \mathbb{N}_0$. Order $\sqsubseteq_{\mathcal{P}}$ over this set is defined by:*

$$(s_0, n_0) \sqsubseteq_{\mathcal{P}} (s_1, n_1) \iff (s_0 \sqsubset_{\mathcal{S}} s_1) \vee ((s_0 = s_1) \wedge (n_0 \geq n_1))$$

The state of the pipeline at a point of the execution of a program records the current progress of all the instructions in the pipeline.

Definition 4. ***(Pipeline state)*** *A pipeline state is described as a subset $C \subseteq \mathcal{I} \rightarrow \mathcal{P}$. Order $\sqsubseteq$ on pipeline states is defined by:*

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

Definition 5. ***(Cycle function)*** *Function $cycle : C \rightarrow C$ describes how instructions progress in the next clock cycle.*

Definition 6. ***(Monotonicity)*** *A pipeline has a monotonic progress if:*

$$\forall c_a, c_b \in C,\ c_a \sqsubseteq c_b \Rightarrow cycle(c_a) \sqsubseteq cycle(c_b)$$

In [11], the authors prove that monotonicity is a sufficient property of a pipeline to guarantee the absence of timing anomalies. In the remainder of the paper, we investigate the impact of using store buffers on the monotonicity of pipelines.

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

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

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

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

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

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

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

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

Figure 3: *cycle*() function for the simplified in-order 5-stage pipeline

4 STORE BUFFERS AND TIMING PREDICTABILITY

In this section, we provide formal models to characterize the effect of store buffers on the two architectural designs discussed in Section 2. In order to keep our results general, we do not provide the complete *cycle* function for a specific pipeline. Instead we focus on the portion of pipeline that is related to the store buffer, and make no particular assumption about the topology of the rest of the pipeline.

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

4.1 Simple in-order pipeline

We define the *cycle* function for the simplified 5-stage pipeline in Figure 3. As pointed out earlier, we only focus on the EX, MEM and WB stages. The previous stages are modeled by an abstract stage called *pre* that initially holds all instructions. The retired instructions go to an abstract stage called *post* after the WB stage. An instruction advances to the next stage when (1) it is ready to advance, and (2) the next stage is guaranteed to be free in the next cycle. The $ready(i)$ function first guarantees that instruction i has been processed in its current stage ($cnt(i) = 0$, where $cnt(i)$ denotes the counter of instruction i, i.e. the number of remaining cycles to complete the processing of i in its current stage). Then, depending on the current stage of i, it checks:

- if i is in *pre*, that i is the oldest instruction in *pre* (instructions enter the pipeline in program order);
- if i is in MEM, that if i is a store, the store buffer is not full.

The $free(s)$ function tells whether stage s will be empty in the next cycle (it is already empty or the instructions it contains will advance to their next stage).

The computation of the counter of remaining processing cycles $c.cnt(i)$ is performed as follows: for any instruction other than a load (and for a load in any stage other than MEM), the counter is decremented if it is positive, and a null counter remains equal to zero. For a load instruction inside the MEM stage, its counter is only decremented if the instruction performs a hit in the cache ($dchit(i)$)[1] or uses the bus in the current cycle ($c.busTaken(i)$ is true).

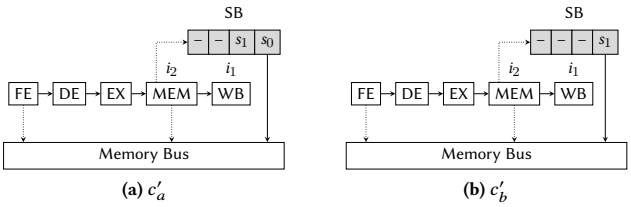


Figure 4: Monotonicity counterexample

Theorem 1. *Store buffers jeopardize the monotonicity of in-order scalar pipelines.*

Proof. We need to exhibit a counterexample where monotonicity is broken, that is to say a sequence of instructions i_1, ..., i_n and two pipeline states c_a and c_b such that $c_a \sqsubseteq c_b$ and $cycle(c_a) \not\sqsubseteq cycle(c_b)$. We start with the states c'_a and c'_b in Figure 4. In this figure, i_2 is a load instruction that requires the bus to access the memory. We assume that in c'_b the s_1 store has been initiated in the previous cycle and will use the bus at least for the next two cycles. As a result, $c'_b.busTaken(i_2)$ is false and $cycle(c'_b).cnt(i_2) = c'_b.cnt(i_2) = memlat_d(i_2)$. Conversely, we assume that s_0 will be completed at the end of the current cycle in c'_a. In the next cycle, s_0 will get out of the store buffer, and i_2 and s_1 will compete for the bus. Let us assume that i_2 gets the bus (i.e. $cycle(c'_a).busTaken(i_2)$). Now, if we rename $c_a = cycle(c'_a)$ and $c_b = cycle(c'_b)$, we have $c_a \sqsubseteq c_b$. From a less formal perspective that considers the stores inside the store buffer, c_a has made less progress than c_b, as s_1 has already started in c_b and not in c_a. Now, since s_1 is assumed

[1] As we will show in Section 5.2, the part regarding $dchit(i)$ is not necessary when modeling a cache whose hit latency is 1 cycle. We keep it here in order to have a general model.

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

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

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

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

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

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

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

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

Figure 5: *cycle*() function for the separate store and load units

to last for at least another cycle, $c_b.busTaken(i_2)$ is false, and the counter for i_2 will remain unchanged in $cycle(c_b)$. On the other hand, $c_a.busTaken(i_2)$ is true, so the counter for i_2 will decrease in $cycle(c_a)$. As a consequence, $cycle(c_a) \not\sqsubseteq cycle(c_b)$. □

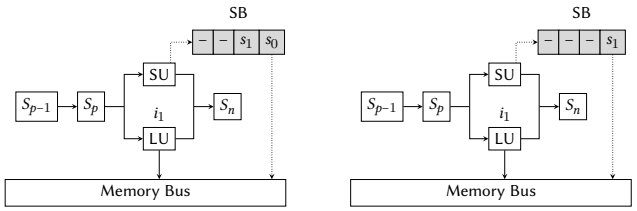
4.2 Separate Store and Load Units

We provide the definition of the *cycle*() function in Figure 5. The main difference with the functions of the previous section concerns the topology of the pipeline that now includes separate SU and LU stages. In order to simplify the model, we consider only load and store instructions in this portion of the pipeline (we assume that other kinds of instructions are being directed to other parallel portions of the pipeline that include their corresponding functional units).

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

Theorem 2. *Store buffers jeopardize the monotonicity of pipelines with separate store and load units.*

Proof. The example of monotonicity violation is just a variation of the one we presented in Section 4.1, adapted to this particular topology. We consider states c'_a and c'_b as in Figure 6. i_1 is a load



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

Figure 6: Monotonicity counterexample on a pipeline with separate store and load units

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

5 ENABLING TIMING-PREDICTABILITY WITH STORE BUFFERS

5.1 Hardware modifications

In order to enforce timing predictability in the presence of store buffers, we propose to add a new gating mechanism that blocks load instructions in the stage before the one that actually performs the load (i.e. before MEM or LU in our examples) as long as the store buffer is not empty or there is a store instruction in MEM or SU. The intuition behind this is to guarantee that the instructions acquire the bus following the program order, so that an access corresponding to an older instruction is always performed before an access corresponding to a younger instruction. In the models of Figures 3 and 5, this modification amounts to:

- adding a $c.sbEmpty()$ predicate that is equal to 1 iff the store buffer and the SU (or MEM) stage are empty in state c;
- adding the following line to the *ready* function using a conjunction: $(c.stg(i) = S_p \Rightarrow (i.kind = load \wedge c.sbEmpty()))$ (in which S_p is replaced by EX in the model of Figure 3).

5.2 Proof of timing anomaly freedom

Using these definitions, we can prove the monotonicity of our pipeline models using the Coq proof assistant.

5.2.1 Progress definition in Coq. Listing 1 shows type definitions used for our proofs. First, we define the stages and instructions. They depend on the pipeline being modeled: the stage definition on line 1 corresponds to the model in Figure 5, and can be replaced by

Listing 1: Stage, instruction, and state definitions in Coq

```
1  Variant stage := Pre | Sp | Lu | Su | Post.
2  (* Variant stage := Pre | Ex | Mem | Wb | Post. *)
3
4  Variant instr :=
5    | Load (memlat : nat) (id : nat)
6    | Store (id : nat)
7    | Other (id : nat).
8
9  (* Common definitions *)
10 Definition iprog := (stage * nat)%type.
11 Definition istate := (instr * iprog)%type.
12 Definition pstate := list istate.
13
14 (* Definition of ⊑_P *)
15 Definition iprog_leb ip1 ip2 :=
16   match ip1, ip2 with
17   | (s1, n1), (s2, n2) ⇒
18       if stage_beq s1 s2 then
19         Nat.leb n2 n1
20       else
21         leb s1 s2
22   end.
23
24 (* Definition of ⊑ *)
25 Definition pstate_leb p1 p2 :=
26   let comb := List.combine p1 p2 in
27   (List.length p1 =? List.length p2) &&
28     List.forallb (fun instr ⇒
29       match instr with
30       | ((_, ip1), (_, ip2)) ⇒ iprog_leb ip1 ip2
31       end) comb.
32
33 (* Compares an instruction across two pstates *)
34 Definition compare_two p1 p2 i :=
35   match List.nth_error p1 i, List.nth_error p2 i with
36   | None, _ | _, None ⇒ false
37   | Some (_, ip1), Some (_, ip2) ⇒ iprog_leb ip1 ip2
38   end.
```

Listing 2: Pipeline model in Coq

```
1  Variant sbStateT := Empty | NotEmpty | Full.
2  Scheme Equality for sbStateT.
3
4  (* Ready function *)
5  Definition ready nextId sbState suUsed (i : istate) :=
6    match i with
7    | (opc, (Pre, 0)) ⇒ idx opc =? nextId
8    | (_, (Su, _)) ⇒ negb (sbStateT_beq sbState Full)
9    | (Load _ _, (Sp, 0)) ⇒
10     sbStateT_beq sbState Empty && (negb suUsed)
11   | (_, (_, cnt)) ⇒ cnt =? 0
12   end.
13
14 (* Free functions *)
15 Definition luSuFree nextId sbState trace suUsed stg :=
16   match get_istate_in_stage trace stg with
17   | Some i ⇒ ready nextId sbState suUsed i
18   | None ⇒ true
19   end.
20
21 Definition spFree nextId sbState trace suUsed :=
22   match get_istate_in_stage trace Sp with
23   | Some (opcode, (stg, lat)) ⇒
24     ready nextId sbState suUsed (opcode, (stg, lat)) &&
25     luSuFree nextId sbState trace suUsed (nstg stg opcode)
26   | None ⇒ true
27   end.
28
29 Definition free nextId sbState trace suUsed (i : istate) :=
30   let (opc, state) := i in
31   let (st, _) := state in
32   match nstg st opc with
33   | Lu ⇒ luSuFree nextId sbState trace suUsed Lu
34   | Su ⇒ luSuFree nextId sbState trace suUsed Su
35   | Sp ⇒ spFree nextId sbState trace suUsed
36   | _ ⇒ true
37   end.
38
39 (* Cycle functions *)
40 Definition cycle_i nextId busTaken sbState trace suUsed i :=
41   match i with
42   | (opc, (st, S n)) ⇒
43     if (negb (stage_beq Lu st)) || busTaken then
44       (opc, (st, n))
45     else
46       i
47   | (opc, (st, 0)) ⇒
48     if ready nextId sbState suUsed i &&
49        free nextId sbState trace suUsed i then
50       let nstg := nstg st opc in
51       (opc, (nstg, get_latency opc nstg))
52     else
53       i
54   end.
55
56 Definition cycle sbState busTaken trace :=
57   let nextId := get_next_id trace in
58   let suUsed := existsb (is_in_stage Su) trace in
59   map (cycle_i nextId busTaken sbState trace suUsed) trace.
```

the commented definition on line 2 for the model in Figure 3. In our proofs, instructions are modeled by a unique integer identifier that reflects their order in the program (line 4). Moreover, each instruction is characterized by its kind (`Load`, `Store` or `Other`). A `Load` instruction also contains the memory latency (a natural number) for the corresponding access (line 5). We then define instruction progress as a couple of a stage and a latency on line 10. Finally, we define a pipeline state as a list of couples composed of an instruction and its progress on lines 11–12.

`iprog_leb`, defined on line 15, takes two `iprogs` (`ip1` and `ip2`), and if they are in the same stage (determined with `stage_beq`), returns true if the latency of `ip2` is lower or equal to that of `ip1`. Otherwise, it compares their stages using `leb`. `stage_beq` is defined automatically with the `Scheme Equality` statement, which creates boolean equality comparators automatically. `leb` is defined manually as an equivalent of the $\sqsubseteq_{\mathcal{S}}$ operator from Definition 2.

`pstate_leb` (line 25) takes two `pstates` (`p1` and `p2`). First, it compares their length, and combines both pipeline states such

that for all i, comb(i) = (p1(i), p2(i)), and then compares the iprogs of each couple. It effectively implements operator $\sqsubseteq$ from Definition 4.

compare_two (line 34) also takes two pstates (p1 and p2) and a natural number (i). Using List.nth_error, a standard Coq function, it tries to get the instructions at position i in both pstates, and compares their iprogs using iprog_leb. If one of the lists does not have i elements, it returns false. Otherwise, it is equivalent to:

$$\texttt{compare\_two p1 p2 i} \iff p_1(i) \sqsubseteq_{\mathcal{P}} p_2(i)$$

These three definitions are pipeline agnostic, relying on the existence of stage, iprog, istate, pstate, stage_beq, and leb.

5.2.2 Pipeline model in Coq. The pipeline model of Figure 5 is represented in Coq using the definitions of Listing 2. For the sake of simplicity we leave out of this listing the definitions of the most simple functions such as nstg and focus instead on the ready, free and cycle functions.

The definition of the ready function (lines 5–12) follows closely the definition of Figure 5. It pattern matches an instruction state:

- if the instruction is still is Pre, it is ready only if its id is the smallest of all instructions still in Pre (line 7)
- if the instruction is in Su, it is ready if the store buffer is not full (line 8)
- otherwise it is ready if the counter of the instruction in its current stage has dropped to 0 (line 11).

We also added the predicate corresponding to our hardware mechanism in line 10: if the instruction is a Load in Sp, it is ready only if the store buffer and the Su stage are empty.

The *free* function from Figure 5 is recursive. This is an issue as Coq requires a proof of termination, most of the time performed automatically through structural induction. To keep things simple, this function is split into three definitions: luSuFree, spFree, and free. The luSuFree function corresponds to the Lu and Su stages. It basically states that the stage given in input is free if there is no instruction in it (line 18), or if the instruction is ready to leave the stage in the next cycle (line 17). This is a simplification of the definition of Figure 5, knowing that in the model the output stage of Lu and Su is Post, which is always free. As the portion of the pipeline that we model is assumed to be managed by a scoreboard, this simplification is valid (instructions leaving the Lu or Su stage can go to the commit stage without delay). We leave for future work the elaboration of additional Coq code to support more complex pipeline layouts. The spFree function manages the "fork" after the Sp stage: the Sp stage is free if there is no instruction currently in it (line 26) or if the instruction is ready to leave, and its next stage (Lu or Su depending on its opcode) is free (line 23). The free function defined in line 29 takes as input an instruction state, pattern matches the next stage for the instruction, and determines if it is free, using the luSuFree and spFree functions and knowing that the Post stage is always free.

Finally, the cycle function is defined in line 56 as the point-wise application of function cycle_i on each instruction state in the current pipeline state (variable trace), at line 59. The cycle_i function implements the $cycle(c)(i)$ and $c.ncnt(i)$ functions of Figure 5:

- if the instruction has a positive counter (it has not yet been fully processed in its current stage), its counter is decremented, provided it is not a Load instruction in the Lu stage (negb (stage_beq Lu st)), or the memory bus is available (busTaken) (line 43). The busTaken boolean is an input to the function that corresponds to the instruction. There is no mention of the $dchit(i)$ predicate here since if the instruction hits the cache, its counter is initialized to 0 when entering the Lu stage, and thus the instruction state cannot be matched to this pattern matching case.
- if the instruction counter is equal to 0, the instruction advances to the next stage in the pipeline if it is ready to advance and the next stage is free (lines 48–49), and remains where it is otherwise.

5.2.3 Proofs in Coq. Listing 3 is an excerpt of the proofs in Coq, in which we consider a sequence of instructions $\mathcal{I} = i_1, i_2, ..., i_n$ processed through a pipeline. The main idea of the proof is to assume two pipeline states c_a and c_b such that $c_a \sqsubseteq c_b$ and any instruction $i \in \mathcal{I}$, and to show that $c_a(i) \sqsubseteq_{\mathcal{P}} c_b(i) \Rightarrow cycle(c_a)(i) \sqsubseteq_{\mathcal{P}} cycle(c_b)(i)$, which is equivalent to $c_a \sqsubseteq c_b \Rightarrow cycle(c_a) \sqsubseteq cycle(c_b)$ according to Definition 4[2].

The initial assumptions are enforced in Coq using *hypotheses*, among which we describe some of the most important ones. First, we declare two pipeline states ca and cb, and assume that ca $\sqsubseteq$ cb using hypothesis Hleb on line 7. We consider that there is an instruction i to execute in both pipeline states (hypotheses Hi, and Hi', lines 10–11). As a shortcut, we designate $c_a(i)$ as ea and $c_b(i)$ as eb (hypotheses Hca and Hcb, lines 8–9). Then, we define ca' and cb' as the respective successors of ca and cb (hypotheses Hca' and Hcb', lines 12 and 13).

The proof handles separately the two initial configuration settings:

- we first prove that, when $c_a(i)$ and $c_b(i)$ have the same progress, $cycle(c_a)(i) \sqsubseteq_{\mathcal{P}} cycle(c_b)(i)$. In Coq, we express this in the following way: when ea = eb, then compare_two ca' cb' i = true. Lemma unmoved_is_monotonic on Line 15 is the proof of this statement.
- then, we prove that the property also holds when $c_a(i) \sqsubset_{\mathcal{P}} c_b(i)$ with lemma moved_is_monotonic (line 23).

Using theorem is_monotonic (line 33), we then deduce that this property holds in all cases, since compare_two ca' cb' i = true when ea is inferior or equal to eb, and since two instrs are comparable (lines 44–45).

We used this strategy for the proofs of timing-anomaly freedom for the two pipeline models of Figures 3 and 5. While theorem is_monotonic is the same for both models, modifications were needed to adapt unmoved_is_monotonic, moved_is_monotonic, and other lemmas that depend on the layout of the core.

Some custom tactics were introduced to improve readability. For the pipeline described in Figure 5 in particular, where LU can only host loads, and SU only stores, a tactic was written to discriminate invalid opcode/stage combinations. Other tactics could be written to improve reusability between the models.

[2]Keeping in mind that this equivalence holds for any pipeline design.

Listing 3: Main theorem in Coq

```
Variable i n n' : nat.
Variable ea eb : istate.
Variable opc : instr.
Variable st st' : stage.
Variable ca capre capost cb cbpre cbpost ca' cb' : pstate.

Hypothesis Hleb : pstate_leb t t' = true.
Hypothesis Hca : ca = capre ++ ea :: capost.
Hypothesis Hcb : cb = cbpre ++ eb :: cbpost.
Hypothesis Hi : i = List.length capre.
Hypothesis Hi' : i = List.length cbpre.
Hypothesis Hca' : ca' = cycle ca.
Hypothesis Hcb' : cb' = cycle cb.

Lemma unmoved_is_monotonic : forall n,
  ea = (opc, (st, n)) →
  (ea = eb →
   (compare_two ca' cb' i = true)).
Proof.
  (* ... *)
Qed.

Lemma moved_is_monotonic : forall n n',
  ea = (opc, (st, n)) →
  (eb = (opc, (st', n')) →
   (istate_leb (st, n) (st', n') = true →
    (ea <> eb →
     (compare_two ca' cb' i = true)))).
Proof.
  (* ... *)
Qed.

Theorem is_monotonic : forall n n',
  ea = (opc, (st, n)) →
  (eb = (opc, (st', n')) →
   (istate_leb (st, n) (st', n') = true →
    (compare_two ca' cb' i = true))).
Proof.
  intros n n' He He' Hsl.

  compare ea eb.
  - now apply (unmoved_is_monotonic n).
  - now apply (moved_is_monotonic n n').
  - repeat decide equality. (* instrs are comparable *)
  - repeat decide equality. (* iprogs are comparable *)
Qed.
```

5.2.4 Key points behind the proofs. We do not display the full Coq scripts in this paper as they would take several pages, but the sources are available at [9]. In the remainder of this section, we present the key points of the proof of monotonicity of the pipeline model described in Figure 5. All the cases listed below were enumerated by the assistant, and therefore cover every situation. According to the model, instructions of kind *other* should not reach S_p, and are therefore omitted from this proof.

Table 1: Coq notations and their equivalent in our formal framework

Coq notation	Meaning
`i`	An arbitrary instruction
`ca` (resp. `cb`)	Pipeline stage c_a (resp. c_b)
`ea` (resp. `eb`)	$c_a(i)$ (resp. $c_b(i)$)
`ca'` (resp. `cb'`)	$cycle(c_a)$ (resp. $cycle(c_b)$)
`compare_two ca cb i`	$c_a(i) \sqsubseteq_{\mathcal{P}} c_b(i)$

When `ea` *and* `eb` *have the same progress:* This lemma is carried out in a case-by-case basis, depending on the stage where `i` can currently reside in `ca` and `cb`.

- Instructions in *post* remain in this stage.
- Loads in `LU` progress if they are ready, otherwise their counter decreases if they can take the bus. If a load in `ca` can take the bus, a load in `cb` is also able to take it.
- Stores in `SU` progress if the store buffer is not full. If `i` is in `SU` in `ca` and in `cb`, the buffer can be full in `cb` only if it is also full in `ca`.
- Instructions in `Sp` progress differently, depending on their opcode and the state of `LU` and `SU`.
  - If the next stage is free in `ca`, and `i` progresses in the next cycle:
    - For loads, this means that there is no pending write in the store buffer and in `SU`. This is also the case in `cb`, so `i` is able to progress in both cases.
    - For stores, this means that the store buffer is not full. This is also the case in `cb`, so `i` progresses in both cases.
  - If the next stage is not free in `ca`, but may be free in `cb`: `i` progresses only in the latter, preserving the order.
- The progress of instructions in `Pre` depends on `Sp`, and whether they are ready (all older instructions have left `Pre`) or not.
  - If `Sp` is empty in `ca`, then it is also empty in `cb`. In both states, `i` can progress.
  - If `Sp` is not empty in `ca` and its instruction cannot progress, or if `i` is not ready in `ca`, then `i` does not progress in `ca'`.

When `ea` *has less progress than* `eb`*:* This proof is done for 4 subcases.

(1) The counter of `i` is equal to 0 in both `ca` and `cb`. This means that, in `ca`, `i` is in an earlier stage than in `cb`, and in the worst case, it can only reach the same stage as in `cb`.
(2) The counter of `i` is higher than 0 in `ca`, but equal to 0 in `cb`. Necessarily, in `ca`, `i` is in `LU`, and its counter may be reduced by 1 in the next cycle. In `cb`, `i` is either in `LU`, in which case it can progress to `Post` at the next cycle, or is already in `Post`, hence `compare_two ca' cb' i = true`.
(3) In `ca`, the counter of `i` is equal to 0, while in `cb`, it is higher than 0. Necessarily, `i` is in `Pre` or in `Sp` in `ca`, and in `LU` in `cb`. In each case, the counter of `i` in `cb` is less or equal than $memlat_d(i)$.
   - if `i` is in `Pre` in `ca`, it is in `Pre` or in `Sp` in `ca'`, so it has less progress than in `cb'`: `compare_two ca' cb' i = true`.

- if i is in Sp in ca, it is in Sp or LU in ca'. In the first case, i has less progress in ca' than in cb'. In the second case, i enters LU so its counter is equal to $memlat_d(i)$ in ca'. As a result, compare_two ca' cb' i = true.

(4) The counter of i is higher than 0 in both ca and cb, and is strictly higher in ca than in cb. Necessarily, both instructions are in LU, and the counter of i in cb is strictly lower than $memlat_d(i)$. This means that in cb, the bus is taken by i (and the corresponding memory request cannot be interrupted). As a consequence, the counter of i will be decremented in cb' and will thus remain lower than the counter of i in ca'. Thus, compare_two ca' cb' i = true.

This proof can be generalized to enable instructions to have a non-zero latency in stages other than LU.

6 DESCRIPTION OF OUR EXPERIMENTAL ARCHITECTURE

We implemented our proposed mechanism in the MINOTAuR RISC-V core [6] to evaluate its impact on performance. MINOTAuR is a 6-stage in-order processor, featuring complex mechanisms such as a scoreboard and parallel functional units. The load/store unit is pipelined: in its second stage, stores are handled by the store unit, and loads by the load unit. The memory hierarchy of MINOTAuR is depicted in Figure 7.

This architecture is a superset of the model described in Figure 5. The data cache acts as an intermediary between the core and the memory bus. The mechanism used to send and remove commands from the store buffer, which is abstracted in the model, is described in the remainder of this section.

6.1 The store buffer

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

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

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

6.2 The data cache

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

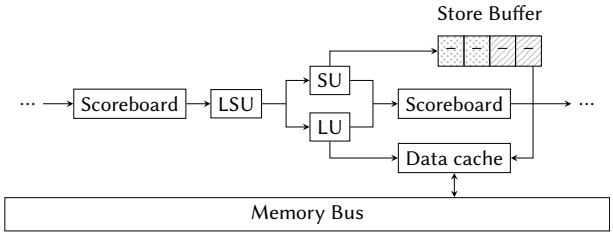


Figure 7: Memory components in the MINOTAuR core

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

7 EXPERIMENTS

In our experiments, we compare the performance of the MINOTAuR core with and without the gating mechanism described in Section 5, to assess the cost of timing predictability regarding the store buffer. Additionally, we also implemented a more aggressive version of our gating mechanism in which load instructions are stalled directly in the LU stage. We believe that this mechanism also enforces the monotonicity of the pipeline. However, since it relies on implementation details of MINOTAuR, formally proving this property would require adding details to the model that would undermine its generality. In the scope of this paper, we focus on generic mechanisms and solutions, and thus leave the specific modeling (and proof) of this particular implementation for future work.

7.1 Experimental setup

The modifications were made to the SystemVerilog description of MINOTAuR, synthesized with Xilinx Vivado 2021.1 for a Xilinx Zynq XC7Z020-1CLG400 on a Digilent Zybo Z7-20 board, with a frequency of 25 MHz, and a memory latency of 11 cycles. All the results presented in this paper correspond to actual measurements performed on the FPGA, running either CoreMark[3] or the TACLe[4] [10] benchmark suite compiled with gcc 10.2.0[5], respectively at optimization level -O3 and -O2. We also ran the TACLe benchmarks at optimization level -O0, to prevent gcc from improving the memory usage patterns, and thus potentially hiding part of the cost of our changes.

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

[3] www.coremark.org
[4] Programs mpeg2 and susan were excluded because they failed to compile (resp. execute) due to memory exhaustion.
[5] https://github.com/riscv-collab/riscv-gnu-toolchain/tree/ed53ae7

Table 2: LUT usage, CoreMark score and performance on the TACLe benchmarks

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

7.2 Results

Table 2 shows that our gating mechanism does not significantly increase the resource usage of the core: 8 LUTs are added to the 17,145 LUTs of the original design.

On average, our gating mechanism results in a loss of performance of 2.07% on the TACLe benchmarks, as reported in Table 3. On individual benchmarks, the performance loss is inconsistent: some benchmarks, such as `bitonic`, are not significantly affected by our gating mechanism, but programs with different memory access patterns (i.e. `md5`, `sha`) are more impacted by this change. A few benchmarks, such as `h264_dec`, are instead significantly faster. We also see that performing the gating in the LU stage yields slightly better average results than performing it in the LSU stage (around 0.5% improvement).

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

8 RELATED WORK

The concept of timing anomalies was first introduced in [16] for processors featuring dynamic scheduling of instructions. Following this publication, two main lines of work have been developed regarding timing anomalies.

The first line of work represents an attempt to formally characterize what timing anomalies are, and to detect their occurrence in processors. In [20], the authors introduce a basic formal characterization of timing anomalies and separate them in two categories: *counter-directive* (also called counter-intuitive in the literature) and *strong impact* (also called amplifying) timing anomalies. A counter-intuitive timing anomaly occurs when a local worst-case event (e.g. a cache miss) does not lead to the WCET. On the other hand, an amplifying timing anomaly occurs in the presence of variable latencies: a local latency variation of δ (e.g. a division takes t+δ time units instead of t units) leads to a global variation of $\Delta > \delta$ (e.g. the program execution time takes T + Δ instead of T). Subsequent works have characterized timing anomalies more finely using either trace-based models [5] or a so-called *canonical pipeline model* [15]. More recently, Binder et al. [2] tried to define a unified formalism to characterize all kinds of timing anomalies, in particular by using the notion of causality [4] and applying it to the TriCore architecture [3].

In the second line of work, researchers build provably timing anomalies-free processor cores. The work that we present in this paper falls within this category. In [11, 12], Hahn et al. introduce SIC: a timing anomaly-free in-order core, as well as a formal modeling and proof framework. Using the same framework, [17] introduced Vicuna, a vector coprocessor that was proven free of timing anomalies. Finally [6, 8] presented MINOTAuR, an open source RISC-V predictable core featuring a restricted degree of out-of-order execution as well as branch prediction mechanisms. However, in these cores, the main focus was always put on the conflicts between the instruction cache and the data cache to use the memory bus. To our knowledge, this is the first time that conflicts related to the store buffer are modeled and dealt with.

9 CONCLUSION

In this paper, we considered store buffers and their impact on the monotonicity of processor pipelines. We first showed that using store buffers breaks the monotonicity of pipelines, both in simple and in more complex pipeline designs.

We then proposed a simple gating mechanism: a load instruction is prevented from entering its memory unit as long as a store is pending in the store buffer. Using the Coq proof assistant, we proved that we achieve monotonicity when this gating mechanism is enabled, both on single-stage and multi-stage memory units.

We then implemented this gating on the MINOTAuR processor, measured its performance impact, and concluded that this impact was negligible on average (around 2% on the TACLe benchmark suite). The changes to the MINOTAuR core presented in this paper, as well as the proofs in Coq, are provided as open-source code [7, 9].

In the future, we plan on generalizing our Coq proofs to more complex architectures (complete pipelines and stages with non-zero latency). Our objective is to write (or to a certain extent generate) proofs that can be reused for different core designs. At the same time, we also plan on designing a predictable core with more complex mechanisms such as a superscalar pipeline and out-of-order execution.

REFERENCES

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

[2] B. Binder, M. Asavoae, B. Ben Hedia, F. Brandner, and M. Jan. 2021. Is This Still Normal? Putting Definitions of Timing Anomalies to the Test. In *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2021, Houston, TX, USA, August 18-20, 2021.* 139–148. https://doi.org/10.1109/RTCSA52859.2021.00024

[3] B. Binder, M. Asavoae, F. Brandner, B. Ben Hedia, and M Jan. 2022. Formal modeling and verification for amplification timing anomalies in the superscalar TriCore architecture. *Int. J. Softw. Tools Technol. Transf.* 24, 3 (2022), 415–440. https://doi.org/10.1007/s10009-022-00655-1

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

[5] F. Cassez, R. Rydhof Hansen, and M. C. Olesen. 2012. What is a Timing Anomaly?. In *12th International Workshop on Worst-Case Execution Time Analysis.* Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1–12. https://doi.org/10.4230/OASIcs.WCET.2012.1

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

Table 3: Overheads for the gating mechanism on TACLe

	Gating at LSU		Gating at LU	
	At -O2	At -O0	At -O2	At -O0
binarysearch	-0.32%	1.79%	2.32%	1.85%
bitcount	7.31%	2.48%	12.30%	3.10%
bitonic	0.00%	3.58%	0.00%	1.57%
bsort	0.00%	3.50%	0.00%	0.00%
complex_updates	0.01%	2.00%	0.07%	0.47%
cosf	0.05%	-0.09%	0.07%	0.17%
countnegative	1.90%	3.11%	6.05%	1.56%
cubic	0.02%	0.85%	0.02%	0.42%
deg2rad	0.00%	-0.34%	0.00%	0.00%
fac	0.00%	0.22%	0.00%	0.00%
fft	2.09%	0.57%	0.95%	0.42%
filterbank	0.00%	0.69%	0.01%	0.01%
fir2dim	-0.40%	0.75%	0.12%	0.03%
iir	-1.12%	9.41%	0.21%	0.09%
insertsort	7.41%	7.47%	1.23%	0.30%
isqrt	1.14%	8.85%	0.19%	0.04%
jfdctint	2.42%	7.64%	0.67%	0.55%
lms	1.07%	0.22%	0.29%	0.09%
ludcmp	0.03%	0.51%	0.06%	0.98%
matrix1	3.08%	1.77%	0.07%	0.01%
md5	12.39%	5.15%	2.28%	1.14%
minver	0.09%	0.68%	0.11%	0.91%
pm	0.55%	2.53%	0.00%	0.67%
prime	-0.11%	-1.07%	0.00%	0.19%
quicksort	-0.99%	10.07%	1.38%	0.69%
rad2deg	0.00%	-0.34%	0.00%	0.00%
recursion	-0.23%	-1.98%	0.00%	0.01%
sha	17.74%	4.60%	12.40%	0.10%
st	0.00%	0.06%	0.00%	0.16%
adpcm_dec	0.02%	2.45%	0.03%	0.05%
adpcm_enc	0.06%	2.46%	0.02%	0.07%
ammunition	-0.06%	2.32%	0.36%	1.90%
anagram	2.94%	4.28%	2.55%	6.38%
audiobeam	-0.02%	1.26%	0.10%	0.17%
cjpeg_transupp	10.41%	7.50%	7.23%	0.07%
cjpeg_wrbmp	7.67%	3.11%	6.44%	0.01%
dijkstra	0.20%	-2.78%	0.22%	0.07%
epic	0.15%	0.13%	0.15%	0.07%
fmref	0.03%	0.16%	0.04%	0.09%
g723_enc	1.13%	2.69%	0.79%	0.94%
gsm_dec	2.00%	5.42%	1.18%	0.24%
gsm_enc	8.39%	6.87%	1.95%	1.27%
h264_dec	-4.61%	2.00%	0.95%	0.10%
huff_dec	1.41%	2.54%	2.88%	0.39%
huff_enc	4.40%	7.15%	0.87%	1.33%
ndes	2.41%	7.77%	3.45%	0.15%
petrinet	0.00%	0.18%	0.00%	0.24%
rijndael_dec	3.87%	4.84%	0.46%	0.17%
rijndael_enc	3.82%	4.74%	0.46%	0.16%
statemate	5.31%	4.83%	5.48%	14.47%
Arithmetic mean	2.07%	2.89%	1.53%	0.88%
Geometric mean	2.00%	2,85%	1.49%	0.85%
Global overhead	0.46%	1.86%	0.32%	1.16%
Minimum	-4.61%	-2.78%	0.00%	0.00%
Maximum	17.74%	10.07%	12.40%	14.47%

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

[8] A. Gruin, T. Carle, C. Rochange, H. Cassé, and P. Sainrat. 2022. MINOTAuR: A Timing Predictable RISC-V Core Featuring Speculative Execution. *IEEE Trans. Comput.* 72, 1 (2022), 183–195. https://doi.org/10.1109/TC.2022.3200000

[9] A. Gruin, T. Carle, C. Rochange, and P. Sainrat. 2023. *Source code for the Coq proofs.* https://gitlab.irit.fr/tacoq/storebuffer

[10] H. Falk et al. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016) (OpenAccess Series in Informatics (OASIcs), Vol. 55)*, Martin Schoeberl (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:10. https://doi.org/10.4230/OASIcs.WCET.2016.2

[11] S. Hahn and J. Reineke. 2018. Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core. In *IEEE Real-Time Systems Symposium (RTSS)*. 469–481. https://doi.org/10.1109/RTSS.2018.00060

[12] S. Hahn and J. Reineke. 2020. Design and analysis of SIC: A provably timing-predictable pipelined processor core. *Real Time Systems* 56, 2 (2020), 207–245. https://doi.org/10.1007/s11241-019-09341-z

[13] S. Hahn, J. Reineke, and R. Wilhelm. 2015. Towards compositionality in execution time analysis: definition and challenges. *ACM SIGBED Review* 12, 1 (2015), 28–36. https://doi.org/10.1145/2752801.2752805

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

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

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

[17] M. Platzer and P. Puschner. 2021. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:18. https://doi.org/10.4230/LIPIcs.ECRTS.2021.1

[18] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. 2006. A Definition and Classification of Timing Anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06) (OpenAccess Series in Informatics (OASIcs), Vol. 4)*, Frank Mueller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. https://doi.org/10.4230/OASIcs.WCET.2006.671

[19] The Coq Development Team. 2022. The Coq Proof Assistant, version 8.15. https://doi.org/10.5281/zenodo.5846982

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