Validation of Processor Timing Models Using Cycle-Accurate Timing Simulators

Alban Gruin
IRIT – Univ. Toulouse 3 – CNRS, France

Thomas Carle
IRIT – Univ. Toulouse 3 – CNRS, France

Christine Rochange
IRIT – Univ. Toulouse 3 – CNRS, France

Pascal Sainrat
IRIT – Univ. Toulouse 3 – CNRS, France

Abstract

We propose a workflow to help find errors in the processor models that are used to prove their timing predictability. Recently, several papers have modeled processor cores using formal models that represent how instructions progress through the pipeline in each execution cycle. However, such models grow with the complexity of the cores and they are built by hand, using a description of the core, usually the HDL-level code. Such a task is error-prone, and verifying that the model actually captures the core's timing behavior is required, otherwise the proofs become useless. Our workflow simulates the execution of benchmark applications using the HDL specification of a core in order to extract timing information as well as other relevant information (e.g. cache miss events, branch mispredictions). This information is used to replay the execution in a simulator of the core timing model, and to determine whether or not the model accurately represents the execution timing of the instructions. To avoid writing the simulator by hand for each new core, or new variation of a core, we developed a compiler that translates the timing model of a core into a C++ program. We evaluated our approach on the open source MINOTAuR core and we show how it enabled us to detect and correct errors in its model.

2012 ACM Subject Classification Hardware → Simulation and emulation; Hardware → Equivalence checking; Hardware → Safety critical systems

Keywords and phrases Processor model, timing predictability, simulator generation

Digital Object Identifier 10.4230/OASIcs.WCET.2023.2

Funding 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.

1 Introduction

Ensuring the schedulability of real-time systems requires knowing the worst-case execution time (WCET) of each critical task. Deriving such a WCET for tasks running on multicore processors is particularly challenging because tasks running in parallel on separate cores may request accesses to shared hardware components (e.g. shared cache, memory bus or controller) at the same time. This makes the WCET of tasks dependent on their execution context and would require a cycle accurate model of the execution of the whole task set, which is untractable in practice. In order to reduce the complexity of the multi-core WCET analysis, the community has adopted the compositional approach [10], in which the WCET of each task is a composition of its worst-case duration in isolation and of a context-related penalty that is computed as part of an interference analysis.

However, this approach can be safely applied only if the cores are not prone to so-called timing anomalies [14] i.e. situations in which a local worst case (e.g. a cache miss) does not lead to the global worst case (i.e. the WCET of the analyzed task). Proving the absence of timing anomalies in a core requires a formal timing model that expresses all the details of the inner workings of the core [11, 8, 13]. This model is usually produced manually from the VHDL or SystemVerilog specification of the core, which is particularly complex, time-consuming and error-prone. As such, the formal model is the weak point of the chain, as the applicability of the proofs relies on the fact that the model strictly reflects the behavior of the processor as it is implemented in hardware.

In this paper, we present a methodology to obtain a better level of confidence on the correctness of such formal models, by simulating the timing behavior of the formal model of the processor using instruction traces obtained from the execution of programs on the target processor. We applied our work on the open source MINOTAuR core [8], a processor derived from the RISC-V CVA6 core [16] which was proven to be timing-anomaly free. Our methodology has allowed us to uncover small mistakes in the formal model of MINOTAuR and to fix them. Although this technique does not guarantee correctness, it allows us to gain a higher confidence in the model. To facilitate the implementation of our methodology, we also propose a model description language and an automatic model simulator generator.

The paper is organized as follows. In Section 2, we present the main lines of our model of the MINOTAuR processor. Our validation workflow is introduced in Section 3 and is evaluated on our model in Section 4. In Section 5, we show how model simulators can be automatically generated from the description of a processor model. We discuss related work in Section 6 and conclude the paper in Section 7.

2 Background

In [8], we introduced MINOTAuR, a 6-stage in-order RISC-V core of moderate complexity. MINOTAuR is a timing-anomaly-free version of the CVA6 core [16]. Its monotonicity was proven using a model similar to the one proposed for the SIC processor [11]. This model is expressed in the logic of predicates. It describes the progression of an instruction in the pipeline of the processor, depending on various factors, such as its kind, memory dependencies, or data dependencies. The model is reproduced on Figure 1.

In the first part, it describes the basic structure of the pipeline: name and order of the stages, latency of an instruction in each stage, next stage of an instruction depending on its kind, etc. In each execution cycle c, an instruction i is in a stage $c.stg(i)$, and has a counter $c.cnt(i)$ that indicates the remaining processing time of the instruction in this stage. The $cycle(c)(i)$ function describes what happens to instruction i at the end of cycle c: either the instruction is ready to advance to the next stage and the next stage is ready to process it, in which case the instruction advances, or the instruction remains in its current stage, in which case its processing counter is decreased by one. When the counter reaches 0, the instruction is considered processed in its current stage.

In the second part of the model, the $c.ready(i)$ predicate describes whether an instruction is ready to advance to the next stage in the next cycle: in the general case its processing counter must be equal to 0 and it must be the oldest instruction in the stage. Depending on the stage the instruction currently resides in, additional constraints may apply. For example, in the issue stage, an instruction is not ready if it has a pending data dependency. Potential branch misprediction is accounted for by the $pwrong(i)$ predicate. We refer the interested reader to the original MINOTAuR paper for a comprehensive description of the model.

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

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

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

$$c.nlat(i) := \begin{cases} memlat_f(i) & : c.nstg(i) = \text{IF} \wedge \neg ichit(i) \\ memlat_d(i) & : (c.nstg(i) = \text{LU} \wedge \neg dchit(i)) \\ & \quad \vee c.nstg(i) = \text{ST} \\ exlat(i) & : c.nstg(i) = \text{DIV} \\ 0 & : otherwise \end{cases} \qquad c.ncnt(i) := \begin{cases} c.cnt(i) - 1 & : c.cnt(i) > 0 \\ 0 & : otherwise \end{cases}$$

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

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

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

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

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

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

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

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

Figure 1 Model of the MINOTAuR core, described in predicate logic, taken from [8].

In the third part, the $c.free(s)$ predicate describes whether a pipeline stage will be free in the next cycle, i.e. whether it can accept a new instruction. As the MINOTAuR core features instructions queues, notably in its issue stage, we also have a feature to count instructions in a stage, and a predicate, $c.slot(s)$, that indicates whether there will be a slot available for an instruction in the next cycle.

Some predicates (e.g. $ichit(i)$, $dep(i, j)$, $pwrong(i)$) and latencies (e.g. $memlat_d(i)$) remain opaque: in order to simplify the model, how their value is obtained is not expressed in the model. For example, $ichit(i)$ is true iff the fetch of instruction i leads to an instruction cache hit. Computing this value for a given instruction sequence would require adding the description of the cache to the model. Instead, the proofs cover both possibilities for the value of $ichit(i)$, and the actual model of the cache is not required.

3 Validation workflow

Our validation workflow relies on a simulator of the formal processor model: it is a C++ implementation of the $cycle(c)(i)$ function from this formal model, and thus it only focuses on the timing aspects of the execution.

Figure 2 (upper part) displays the validation workflow. We use a bit-accurate cycle-accurate simulator generated from the Verilog description of the processor and simulate the execution of a benchmark application. For this simulation, we extract the information corresponding to the opaque predicates in our model, allowing us to replay the execution in our simulator. We obtain the following set of traces: reads to the insrtuction cache, reads to the data cache, writes to the data cache, divisions, and finally, committed instructions. These traces are obtained by adding probes to the SystemVerilog design of the core[1].

The execution is then replayed in the model simulator, that extracts the predicate and latency values from the traces, and generates itself a trace of committed instructions.

Finally, we compare the commit traces from the SystemVerilog simulator and from the model simulator: if they are identical, that means that the model accurately describes the timing behavior of the core for this benchmark. Otherwise, it indicates that there is an error in the model. In that case, the trace can help in narrowing down the search of the error.

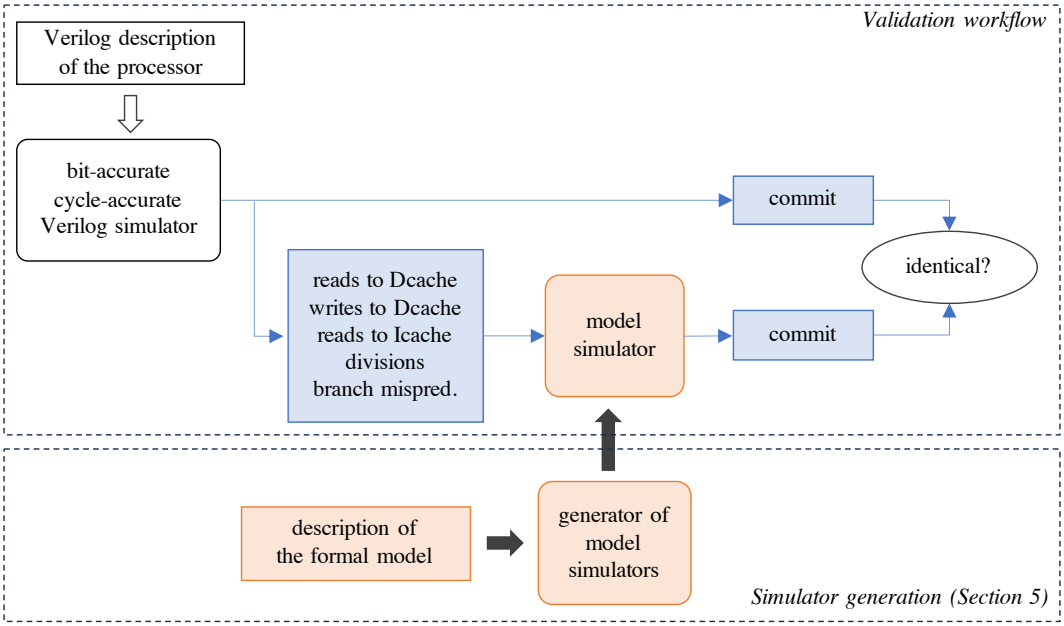

Figure 2 Overview of our validation workflow.

Traces formats

The traces are stored in a simple, plain-text format, with one event (i.e. one cache access, or one division, or one branch misprediction) per line.

[1] For cores whose SystemVerilog design is unavailable, we are currently looking at the feasibility of using a hardware probe e.g. Lauterbach debugging probe.

Instruction cache reads trace. An event consists of 5 fields: whether the request is valid or not (i.e. it has not been cancelled by the frontend due to a branch misprediction), start cycle, read address, end cycle, and instruction binary code. The address of the instruction allows its identification all along its progression in the pipeline, and the binary code is necessary to send the instruction to the correct functional unit during the simulation, as well as to track instruction dependencies (predicate $dep(i, j)$). The start and end cycles determine the $ichit(i)$ predicate and the $memlat_f(i)$ latency. Listing 1 shows 3 lines of an instruction cache trace.

Data cache reads trace. An event contains 3 fields: validity of the request, start and end cycles. This is enough to determine the $dchit(i)$ predicate and the $memlat_d(i)$ latency. Other information, such as the address of the access, are not needed for the model simulation.

Data cache writes and divisions traces. They both have 2 fields per line: start and end cycle of the operation. They determine the $dchit(i)$ predicate and $memlat_d(i)$ and $exlat_d(i)$ latencies.

Branch misprediction trace. An event only contains the cycle at which a branch is determined to be mispredicted by the ALU. This is used to set $pwrong(i)$ for all the younger instructions residing in the pipeline at the time the branch reaches the ALU.

Commit trace. In MINOTAuR, a trace of committed instructions is written by default when simulating. It contains a lot of information, such as the commit time and cycle, the instruction address, the opcode and the decoded instruction, or register values. For our purpose, we only need to extract the commit cycle and the address of the instruction. Hence, the commit traces written by our model simulator contain only these two fields.

Listing 1 Extract of the instruction cache reads trace for the CoreMark benchmark. From left to right: validity, start cycle, read address, end cycle, instruction binary code.

```
1 1       282 00000810       286 7b241073
2 1       287 00000814       291 7b351073
3 1       292 00000818       296 00000517
```

4 Evaluation

In this section, we discuss the results obtained by applying our methodology to the processor, i.e., what issues we found in the model, and the limitations of the methodology.

We began by implementing new tracers to the description of the core (in addition to the existing commit tracer) in order to generate all the aforementioned traces. We then ran the CoreMark benchmark and the full TACLe benchmark suite [7] under QuestaSim[2], a cycle-accurate SystemVerilog simulator, to obtain the traces required by the model simulator.

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

4.1 Issues found in MINOTAuR's model and solutions

During our experiments, we found issues in the formal model of MINOTAuR, sometimes related to confusion or misunderstanding of the behavior or structure of the pipeline, and sometimes due to a misinterpretation of the formal logic used in the model. We present 3 of them in the remainder of this section.

Issue 1. One of the most important issues we found using our validation workflow, was related to data dependencies. In function $c.ready(j)$, our model states that an instruction is ready (i.e. can progress) when it has no pending data dependency, or when the instruction it depends on is in or after the commit stage. While this definition is sufficient for read-after-write (RaW) dependencies, it does not cover most write-after-write (WaW) dependencies. Actually, most RaW dependencies can be resolved as soon as the oldest instruction has completed its execution and reached stage CO: the result will be forwarded to the newer instruction. One exception to this principle are CSR[3] read instructions: the read is done only when the instruction is committed. In the case of WaW dependencies, the newer instruction has to wait for the oldest instruction to be committed before it can be issued. To solve this, we replaced the $dep()$ predicate with two predicates, one for WaW hazards, and another for RaW hazards (resp. $dep_{WaW}()$ and $dep_{RaW}()$). The dependency check becomes:

$$\forall j < i \,.\, (dep_{WaW}(i,\, j) \Rightarrow c.stg(j) \sqsupset_{\mathcal{S}} \text{CO})$$
$$\wedge\, (dep_{RaW}(i,\, j) \Rightarrow ((opc(j) = csr \wedge c.stg(j) \sqsupset_{\mathcal{S}} \text{CO}) \vee (c.stg(j) \sqsupseteq_{\mathcal{S}} \text{CO})))$$

Issue 2. We also found an inconsistency related to the data cache in the LU stage. The Verilog simulator shows that, whenever a cache miss occurs in the LU, no instruction is processed in the cycle that follows the end of the access. This was not represented in the model, and to account for this, we changed the definition of the $c.free()$ function. The relevant part of the model is the following:

$$s \in \{\text{PC, ID, DIV, LU, ST}\} \wedge ((\neg\exists j \,.\, c.stg(j) = s)$$
$$\vee\, (\exists j \,.\, c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j))))$$

which states that the LU is free if there is no instruction in the stage, or if the instruction in the LU exits the stage in the next cycle. Instead, the stage can be special cased to the following:

$$s = \text{LU} \wedge ((\neg\exists j \,.\, c.stg(j) = \text{LU})$$
$$\vee\, (\exists j \,.\, c.stg(j) = \text{LU} \wedge c.ready(j) \wedge c.free(c.nstg(j)) \wedge dchit(j)))$$

The relevant change is highlighted. It means that, if there is an instruction in the LU, the stage will be free if the current instruction is a hit, but not if it is a miss.

Issue 3. Another inconsistency concerns the behavior of instructions in the issue stage: the model specifies that all instructions, except loads, stores and atomics, will not be issued in the presence of an uncommitted CSR instruction. In reality, this does not happen (this error was due to misunderstanding when reading the SystemVerilog code), and the part of the expression that is highlighted below can be removed:

$$c.stg(i) = \text{IS} \Rightarrow (opc(i) \notin \{load,\, store,\, atomic\} \Rightarrow \neg c.pending(i,\, csr))$$
$$\wedge\, (opc(i) \in \{mul,\, div\} \Rightarrow \neg c.pending(i,\, div)) \wedge (\forall j < i \,.\, dep(i,\, j))$$

[3] Control and Status Register

Validation summary. After correcting the model, we were able to run all our benchmarks and found no difference between the commit trace generated by the Verilog simulator and the one obtained by our model simulator.

4.2 Limitations of our methodology

Our validation workflow is based on testing and thus does not provide any guarantee that the chosen benchmarks cover all the possible errors in the model. However, it helped us find and correct a few errors in our model, and thus gain confidence in the revised model.

Additionally, we emphasize that applying our validation workflow with the same benchmarks as those used for performance evaluation purposes guarantees (once all the discrepancies have been fixed) that the performance results have been obtained with a model that reflects the processor's behavior accurately.

5 Automatic generation of timing simulators

One major shortcoming of our method is having to write a simulator that corresponds to the model of the core: this task is error-prone and must be done each time a new core is considered. To ease up the transition from the formal model described in predicate logic, like the one in Figure 1, to an efficient C++ simulator, we designed a domain-specific description language that stays as close as possible to the predicate logic formulas of the model, as well as a compiler written in OCaml. Listing 6 gives the formal definition of our language in EBNF.

It is a functional language similar to both the logic language used in the SIC [11] and MINOTAuR [8] models, and, to some extent, to OCaml. It features some basic data types (integers, booleans, lists, and tuples), with the possibility to define custom enumerations, and optionally, to define an order on the elements of the enumeration. Basic constructs such as simple pattern matching are also available. The types of variables and functions are inferred by the compiler, using a Hindley-Milner type system.

In the remainder of this section, we present key features of our language, using examples taken from the MINOTAuR model on Figure 1, and later explain the compilation process in more details.

5.1 Description of the language

Our language allows the definition of custom enumerations with the `set` or `type` keywords, as well as partial orders on their elements. This can be used to reproduce the definition of the pipeline and instruction kinds in the beginning in our model:

Listing 2 Declaration of the stages and opcodes of the MINOTAuR core.

```
set stage = | Pre | PC | IF | ID | IS | ALU | MUL1 | MUL2
  | DIV | LSU | LU | SU | CSR | CO | ST | Post

order stage as s = Pre < PC < IF < ID < IS < {ALU, MUL1, LSU, CSR, DIV} <
  {MUL2, LU, SU} < CO < ST < Post

type opcode = | Nop | Alu | Mul | Div | Load | Store
  | Atomic | Branch | Csr | Fence | FenceI | Unknown
```

The `order` keyword marks the beginning of the declaration, and the `as` keyword is used to define a suffix for the `<`, `<=`, `>`, and `>=` operators. Here, to compare two stages, one will have to use the `<s`, `<=s`, `>s`, and `>=s` operators, respectively. Elements between curly braces

are given the same level: `IS <s ALU` and `IS <s LSU` are true, but `ALU <s LSU` and `LSU <s ALU` are false, for instance. But, even though `ALU` and `LSU` are different, `ALU <=s LSU` and `LSU <=s ALU` are true.

One can declare variables and functions with the `let` keyword in our language. They can be recursive and co-recursive, as this is required to implement the $c.free(s)$ and $c.slot(s)$ predicates. The user must take care of providing a base case for them. In our case, a recursive call is made to $c.free(c.nstg(j))$, which will eventually be equal to $c.free(post)$. As an example, here is our implementation of the $c.ready(i)$ function:

Listing 3 Implementation of the $c.ready(i)$ function in our language.

```
1  let ready(opc, limit c, i, pwrong) =
2    (stg(c, i) <> Pre /\ !pending(opc, c, i, Branch) /\ pwrong)
3    \/ (cnt(c, i) = 0 /\ isnext(c, stg(c, i), i) /\
4        (stg(c, i) = IS ->
5          (opc[i] in {Mul, Div} -> !pending(opc, c, i, Div))
6          /\ (forall j in c, (j < i -> !dep(opc, c, i, j))))
7        /\ (stg(c, i) = LSU ->
8            (opc[i] in {Store, Atomic} /\ !pending(opc, c, i, Atomic))
9            \/ (opc[i] = Load /\ !pending(opc, c, i, Atomic))))
```

The syntax of the language is very similar to the predicates logic used by the model of MINOTAuR. This example demonstrates multiple features in our language: the multiple comparators and logic connectors, and lists.

Lists are essential in our language to represent traces. Here, `opc` and `c` are lists, used to represent the opcode and the state of an instruction, respectively. To access a specific element, one can use the `[]` operator, as seen on Line 8. It is not advised to use it with a constant or an *ad-hoc* variable, as it may be out-of-bounds. Instead, one can use the `forall`, `exists`, and `#{}` constructs to scan lists. The indices they generate are guaranteed to be valid. Table 1 lists the builtin functions working on lists, as well as their logic equivalent.

Lists exists in two kinds: "normal" lists, and "limited" lists. This distinction exists to allow the code generator to specify bounds for the `forall`, `exists`, and count functions. This allows to load only parts of the traces in memory if the generator elects to do so. The exact bounds are hidden and cannot be manipulated in our language. The only way to obtain a "limited" list is by adding an annotation on a parameter, like on `c` in the example above.

The bounds of a "limited" list cannot be manipulated directly from our language.

It is possible to downcast a "limited" list to a regular list, in which case the bounds will be dropped. It is also possible to upcast a regular list to a "limited" list. In this case, the bounds will cover the whole list.

Table 1 Builtin functions for lists.

Construct	Logic equivalent
`c[i]`, `index(c, i)`	$c[i]$
`forall i in c, (...)`	$\forall i \ldots$
`exists i in c, (...)`	$\exists i \ldots$
`#{j in c | ...}`	$\#\{j \mid \ldots\}$

5.2 Compilation of the model

We developed a compiler for our language, which takes care of type checking and code generation in C++. This compiler only generates code for the portions of the model that vary with each processor (the ready, free, nstg, lstg and slot functions). The high-level functions (cycle, nlat) are hard-coded in a template file that is used for all processors, as well as the I/O functions that read the traces, decode the instructions (find their kind, latencies and dependencies) and set the opaque predicate values.

In our language, all constructs are expressions, which is not the case of conditional blocks and for loops in C++. Thus, boolean expressions, like in Listing 3, may be translated as pure expressions or as a sequence of conditional blocks when loop constructs are required. Loop constructs are generated when using the `forall`, `exists`, or `#{}` list builtins, as shown on Listing 4.

Listing 4 Example of translation of the `forall`, `exists`, and `#{}` constructs.

```
// forall j in c, stg(c, j) = s -> j < i
bool tmp0 {true};
for (unsigned int j {0}; j < c.size(); ++j)
    tmp0 = tmp0 && (j < i || !(stg(c, j) == s));

// exists j in c, stg(c, j) = s -> j < i
bool tmp1 {false};
for (unsigned int j {0}; j < c.size(); ++j)
    tmp1 = tmp1 || (j < i || !(stg(c, j) == s));

// #{j in c | stg(c, j) = s -> j < i}
unsigned int tmp2 {0};
for (unsigned int j {0}; j < c.size(); ++j) {
    if (j < i || !(stg(c, j) == s))
        ++tmp2;
}
```

Notice that the `j` index has a value ranging from 0 to the size of the list minus 1. This is the behavior for regular lists. For limited lists, the index would have a value within the bounds. When a function takes a limited list as a parameter, the bounds are added automatically by the compiler in the prototype and at each call site. The prototype of the `ready` function is generated as shown on Listing 5. The compiler does not modify the bounds, thus it is up to the interface between the generated code and the rest of the simulator to handle them.

Listing 5 C++ prototype of the `ready` function.

```
bool ready(opcode opc, std::vector<stage, unsigned int> c, unsigned int
    ↪ c_start, unsigned int c_end, unsigned int i, bool pwrong);
```

As seen on Listing 5, lists are compiled to standard C++ vectors, which requires to load the entire trace in memory. This works for the majority of programs of the TACLe benchmark suite, but some, such as `ammunition`, were so big that their traces did not fit in the memory of our test machine. To alleviate this problem, our simulator loads chunks of the trace files as they are needed through a special vector type. To benefit from this mechanism, the generated code has to be modified manually for now.

Sets (resp. orders), as shown in Listing 2, are compiled to enumerations (resp. operator overload) in C++. When an order is defined for a set, a function converting each element to an integer is generated. The lowest element (e.g. `Pre`) is assigned the value 0, with this value growing for each level. The overloaded operators then convert each element to an integer, and compare the results.

6 Related Work

Various kinds of formal models of the timing behavior of processors have been proposed in the literature. These models are designed to support the estimation of worst-case execution times [12, 6, 2, 9] or the proof of properties, such as the absence of timing anomalies [11, 4, 1]. They describe how an instruction or a code sequence flows through the pipeline using so-called execution graphs [12, 2], timed automata [6, 9], a transition system [4] or a set of logic predicates [11]. These models are usually designed by hand, either from the processor user manual or from its HDL description.

A few papers consider automatically deriving the processor's model from HDL code [15, 3] to alleviate the risk of errors in the model but they only provide preliminary solutions.

In [5], the authors model the functional and timing behavior of simple processor using the L3 domain specific language, translate it in a HOL4 version and confront it to the execution on the real processor of short code snippets. The processor we consider in this paper implements complex mechanisms (e.g. speculative execution) that are not addressed in their paper.

7 Conclusion and Future Work

We introduced a workflow to validate that a timing model of a processor corresponds to the actual execution timing on the real processor. This workflow is based on a simulator of the model that replays traces obtained by executing (or simulating) the execution of a benchmark on the actual processor. Since the model focuses only on the timing aspects of the core, so does our simulator. This allows the simulator to be very simple, compared to a functional simulator. By comparing the original execution trace to the one obtained with the model simulator, we can detect errors in the model and correct them. We applied this methodology to the model of the open-source MINOTAuR core and were able to find and correct several issues in the model, using the CoreMark and TACLe benches. In order to facilitate the use of this workflow, we also presented a compiler that automatically generates the model simulator from a language that is very close to the predicate language in which the cores are described.

Being based on testing, our workflow does not provide a guarantee that all mistakes have been corrected in a model, but it still allows to increase the confidence one can have in a given model. As part of future work, we envision to extend our workflow to more complex cores featuring out-of-order execution, and to use our description language to automatically generate proofs of the absence of timing anomalies in Coq. We could thus generate proofs and simulators from the exact same model.

References

1 Mihail Asavoae, Belgacem Ben Hedia, and Mathieu Jan. Formal executable models for automatic detection of timing anomalies. In Florian Brandner, editor, *18th International Workshop on Worst-Case Execution Time Analysis,WCET*, 2018.

2 Zhenyu Bai, Hugues Cassé, Thomas Carle, and Christine Rochange. Computing execution times with execution decision diagrams in the presence of out-of-order resources. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

3 Samira Ait Bensaid, Mihail Asavoae, Farhat Thabet, and Mathieu Jan. Work in progress: Automatic construction of pipeline datapaths from high-level HDL code. In *28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

4 Benjamin Binder, Mihail Asavoae, Belgacem Ben Hedia, Florian Brandner, and Mathieu Jan. 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.

5 Brian Campbell and Ian Stark. Randomised testing of a microprocessor model using SMT-solver state generation. In *Formal Methods for Industrial Critical Systems*, 2014.

6 Franck Cassez, Pablo Gonzalez de Aledo, and Peter Gjøl Jensen. Wuppaal: Computation of worst-case execution-time for binary programs with uppaal. In *Models, Algorithms, Logics and Tools*, pages 560–577. Springer, 2017.

7 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, pages 2:1–2:10, 2016.

8 Alban Gruin, Thomas Carle, Christine Rochange, Hugues Cassé, and Pascal Sainrat. MINOTAuR: A timing predictable RISC-V core featuring speculative execution. *IEEE Transactions on Computers*, 72(1):183–195, 2022.

9 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

10 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th international conference on real-time networks and systems*, pages 299–308, 2016.

11 Sebastian Hahn and Jan Reineke. Design and analysis of SIC: a provably timing-predictable pipelined processor core. *Real-Time Systems*, 56(2):207–245, 2020.

12 Xianfeng Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *25th IEEE International Real-Time Systems Symposium*, 2004.

13 Michael Platzer and Peter Puschner. Vicuna: a timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

14 Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

15 Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for WCET analysis. *SIGPLAN Not.*, 45(4):67–76, April 2010.

16 Florian Zaruba and Luca 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*, 27(11):2629–2640, 2019.

A Appendix

A.1 Formal definition of our language

Listing 6 Formal definition of our language in EBNF.

```
digit = "0" | ... | "9";
letter = "a" | ... | "z";
id = letter, {"a" | ... | "z" | "0" | ... | "9" | "_" | "-"};
number = ["-" | "+"], {digit};
subset = "{", {id, "|"}, id, "}";

description = {type_declaration | order_declaration |
    ↪ function_declaration};
```

```
type_declaration = ("type" | "set"), id, "=", ["|"], {id, "|"}, id;
order_declaration = "order", id, "as", id, "=", {(id, subset), "<"}, (id, subset);

param = id | ("limit" id);
param_list = {param, ","}, param;
function_contents = id, ["(", param_list, ")"], "=", expression;
function_declaration = "let", ["rec"], function_contents, {"and", function_contents};

comparison = expression, ("/\" | "\/" | "->" | "=" | "<>" | (("<" | ">"), ["="], [id])), expression;
inside = expression, ["not"], "in", subset;
negation = ("!" | "not"), expression;
forall = ("forall" | "exists"), id, "in", expression, ",", "(", expression, ")";
count = "#{", id, "in", expression, "|", expression, "}";
list_access = expression, "[", expression, "]";
tuple = "(", {expression, ","}, expression, ")";
priority = "(", expression, ")";
funcall = expression, "(", [{["limited"], expression, ","}, ["limited"], expression], ")";
immediate = id | "true" | "false" | number | tuple;
match = "match", expression, "with", {"|", immediate, "->", expression}, ["|", "_", "->, expression], "end";
if = "if", expression, "then", expression, "else", expression;
expression = function_declaration | comparison | inside | negation | forall | count | list_access | priority | funcall | immediate | match | if;
```