Validation of processor timing models using cycle-accurate timing simulators

Alban Gruin Thomas Carle Christine Rochange Pascal Sainrat

July 11th, 2023

Irit, Univ. Toulouse III Paul Sabatier, CNRS

Motivation

The need for precise and accurate timing models

- We require accurate models for WCET analysis and timing anomalies detection

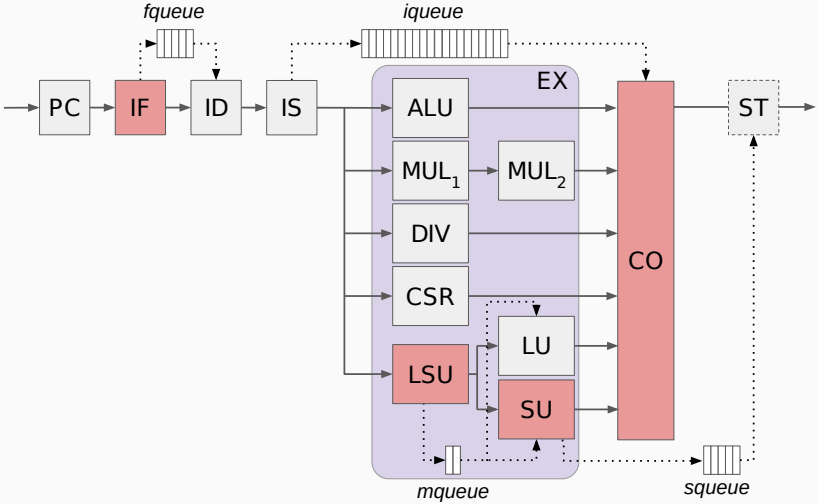
Timing models

Formal pipeline models, based on predicate logic

- Introduced in SIC[1], reused in MINOTAuR[2] and Vicuna[3].
- Based on instruction progress in the pipeline
  - Instructions are associated to a stage and a latency.
  - *cycle*(c) function to get the next pipeline state.
- These models allow proofs on the timing behavior of the processor
  - Timing-anomaly free processors!

[1] Hahn and Reineke, 2018
[2] Gruin, Carle, Cassé, and Rochange, 2021
[3] Platzer and Puschner, 2021

The Ariane/CVA6 (and MINOTAuR) core



Example of timing models: (excerpt from) the MINOTAuR core i

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

$$
\begin{aligned}
&c.free(s) := s \in \{\textsc{alu}, \textsc{mul}_1, \textsc{csr}, \textsc{mul}_2, \textsc{co}, post\} \\
&\quad \vee (s \in \{\textsc{if}, \textsc{is}, \textsc{lsu}, \textsc{su}\} \wedge c.slot(s)) \\
&\quad \vee (s \in \{\textsc{pc}, \textsc{id}, \textsc{div}, \textsc{lu}, \textsc{st}\} \wedge ((\neg \exists j \,.\, c.stg(j) = s) \vee (\exists j \,.\, c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j))))) \\
&\quad \vee (\exists i \,.\, c.stg(i) = s \wedge pwrong(i) \wedge \neg c.pending(i, branch))
\end{aligned}
$$

Example of timing models: (excerpt from) the MINOTAuR core ii

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

Example of timing models: (excerpt from) the MINOTAuR core iii

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

Example of timing models: (excerpt from) the MINOTAuR core iv

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

$$
\begin{aligned}
&c.free(s) := s \in \{\textsc{alu}, \textsc{mul}_1, \textsc{csr}, \textsc{mul}_2, \textsc{co}, post\} \\
&\qquad \vee (s \in \{\textsc{if}, \textsc{is}, \textsc{lsu}, \textsc{su}\} \wedge c.slot(s)) \\
&\qquad \vee (s \in \{\textsc{pc}, \textsc{id}, \textsc{div}, \textsc{lu}, \textsc{st}\} \wedge ((\neg\exists j \,.\, c.stg(j) = s) \vee (\exists j \,.\, c.stg(j) = s \wedge c.ready(j) \wedge c.free(c.nstg(j))))) \\
&\qquad \vee (\exists i \,.\, c.stg(i) = s \wedge pwrong(i) \wedge \neg c.pending(i, branch))
\end{aligned}
$$

Issues with timing models

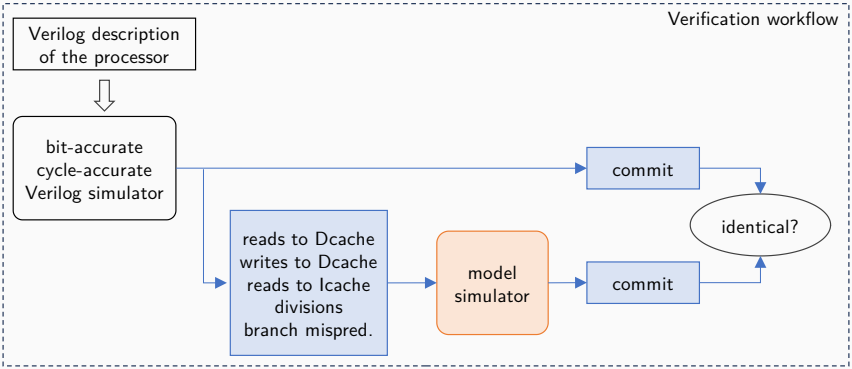
- These models can be tedious to write, and may not be correct wrt. the actual core.
- Hardware descriptions are complex.
  - MINOTAuR $\rightarrow$ ~75.000 lines of SystemVerilog...
- Datasheets are imprecise at best.
- The whole process is not very robust.

Contributions

Validation methodology based on simulation, with a test approach

- Simulation infrastructure for processor models
- Description language for formal processor models
- Simulator compiler

Validation workflow overview



Traces

Trace kind	Contents
Icache	Addresses, opcodes, timings, cancellations of accesses
Dcache reads	Timings and cancellations
Dcache writes, divisions	Timings
Control flow	Cycles at which a misprediction happens
Commit	Address, commit cycle

Experimental evaluation on the MINOTAuR processor

- Tried our workflow on the TACLe benchmark suite and CoreMark.
  - Took 2 days to generate all traces from the processor
  - Validated our model against the benchmarks in ~1h
- Found several issues.
- After fixing the model, the simulator generates the same commit trace as the processor.

First issue, related to data dependencies

WaW and RaW data dependencies are handled differently

- In the model: $\forall \mathrm{i} < \mathrm{j} \,.\, \mathit{dep}(\mathrm{i}, \mathrm{j}) \Rightarrow \mathrm{c}.\mathit{stg}(\mathrm{j}) \sqsupseteq_{\mathcal{S}} \mathrm{co}$
  - I.e. if there is a data dependency between instructions i and j, it will be resolved when the older instruction has completed its execution.
  - True for RaW hazards, not for WaW hazards: writes must be committed before reusing a register.
- Fix:

$$\forall \mathrm{j} < \mathrm{i} \,.\, (\mathit{dep}_{\mathit{WaW}}(\mathrm{i}, \mathrm{j}) \Rightarrow \mathrm{c}.\mathit{stg}(\mathrm{j}) \sqsupset_{\mathcal{S}} \mathrm{co}) \\ \wedge (\mathit{dep}_{\mathit{RaW}}(\mathrm{i}, \mathrm{j}) \Rightarrow ((\mathit{opc}(\mathrm{j}) = \mathit{csr} \wedge \mathrm{c}.\mathit{stg}(\mathrm{j}) \sqsupset_{\mathcal{S}} \mathrm{co}) \vee (\mathrm{c}.\mathit{stg}(\mathrm{j}) \sqsupseteq_{\mathcal{S}} \mathrm{co})))$$

Second issue, related to the load unit

The LU stalls for one cycle after a cache miss

- In the model:

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

- Fix: create a special case for the LU, taking the cache hit into account.

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

Third issue, related to CSRs

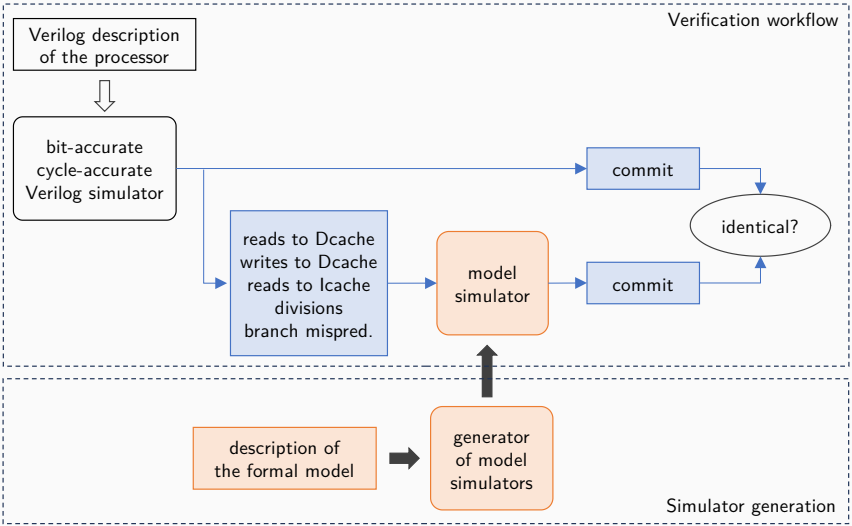
CSR do not prevent arithmetic instructions to be issued

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

(Semi-)automatic generation of timing simulators

- Going from predicate logic to a reasonably fast programming language manually (eg. C++) is also error-prone and time consuming.
- We designed a special-purpose description language to encode predicate logic.
  - Syntax close to OCaml and to the predicate logic used.
  - Functional (predicates used do not mutate anything).
  - Compiled to C++.

Simulator generation workflow overview



Formal processor model description language

- Implements enough constructs to implement MINOTAuR's model.
- Data types (integers, lists, tuples, user-defined enumerations).
- Partial orders on user-defined enumerations.
- User-defined functions and recursive functions.
- Types are inferred by the compiler.
- Sufficient for MINOTAuR (and SIC).

Example

```
set stage = | Pre | IF | ID | IS | ALU | LSU | CO | Post
order stage as s = Pre < IF < ID < IS < {ALU, LSU} < CO < Post
```

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

Compilation to C++

```
// 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;
}
```

- In our language, everything is an expression (not the case in C++).
- Bounds for limited lists, provided by the caller.

Integration with the simulation infrastructure

Building blocks are provided

- Trace readers, using lazy-loading if traces do not fit into working memory.
- Template for the `cycle()` function.
  - It must be able to compute bounds for traces.

Conclusion

- We introduced a workflow to validate processor timing models
  - Uses actual execution traces (obtained eg. using a cycle-accurate simulator)
  - Replays instruction traces and compares the result and commit trace
  - Developed a description language to simplify the transcription process
- Applied it on an existing, timing-predictable processor (MINOTAuR)
  - Found and fixed several issues in the model
  - Our model now conforms to the actual MINOTAuR core, at least on the benchmarks we used
- Future work
  - Automatic Coq generation
  - More complex models (OoO)

Thank you!

A. Gruin, T. Carle, H. Cassé, and C. Rochange. Speculative execution and timing predictability in an open source RISC-V core. In *IEEE Real-Time Systems Symposium (RTSS)*, 2021.

S. Hahn and J. Reineke. Design and analysis of SIC: A provably timing-predictable pipelined processor core. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

M. Platzer and P. Puschner. Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, 2021.