Provably Timing-Predictable Microarchitectures

Jan Reineke

joint work with Sebastian Hahn and Johannes Kahlen
**Context: Hard Real-Time Systems**

Airbag
Reaction in < 10 ms

Embedded software must
- deliver **correct** control signals,
- within **fixed** time bounds.
Timing Analysis Problem

// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i]*b[j-i];
    // Notify listeners.
    notify(x[i]);
}

Embedded Software + ? Microarchitecture = Timing Requirements

Precision-Timed (PRET) Machines – p. 11/19
What does the execution time depend on?

- The input, determining which path is taken through the program.
What does the execution time depend on?

- The input, determining which path is taken through the program.
- The **state of the hardware platform**:
  - Due to caches, pipelining, speculation, etc.

![Diagram of a computer system](image)
Influence of Microarchitectural State

\[ x = a + b; \]

LOAD r2, _a
LOAD r1, _b
ADD r3, r2, r1

PowerPC 755

Execution Time (Clock Cycles)

<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Cycles</td>
<td>50</td>
<td>350</td>
</tr>
</tbody>
</table>
What does the execution time depend on?

- The input, determining which path is taken through the program.
- The state of the hardware platform:
  - Due to caches, pipelining, speculation, etc.
- Interference from the environment:
  - External interference as seen from the analyzed task on shared busses, caches, memory.
Radojkovic et al. (ACM TACO, 2012) on Intel Atom and Intel Core 2 Quad:

up to 14x slow-down due to interference on shared L2 cache and memory controller
Two Schools of Thought

1. Predictable = \( \frac{WCET}{BCET} \) is small \( \approx \) deterministic timing

2. Predictable = WCET can be efficiently approximated
Timing Anomalies

Analysis state representing set of concrete hardware states.

Cache Miss = Local Worst Case

Nondeterminism due to uncertainty about hardware state.

Cache Hit

leads to

Global Worst Case

Timing Anomalies in Dynamically Scheduled Microprocessors
T. Lundqvist, P. Stenström – RTSS 1999
Timing Anomalies: Example

Scheduling Anomaly

Bounds on multiprocessing timing anomalies
(http://epubs.siam.org/doi/abs/10.1137/0117039)
Timing Compositionality: By Example

Timing Compositionality =
Ability to simply sum up timing contributions by different components

Implicitly or explicitly assumed by (almost) all approaches to timing analysis for multi cores and cache-related preemption delays (CRPD).
Timing Compositionality: Benefit

How does compositionality help?

Efficiency of microarchitectural analysis

Integrated

Compositional
Textbook in-order pipeline + LRU caches

<table>
<thead>
<tr>
<th>Fetch (IF)</th>
<th>I-cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode (ID)</td>
<td>Memory</td>
</tr>
<tr>
<td>Execute (EX)</td>
<td></td>
</tr>
<tr>
<td>Memory (MEM)</td>
<td>D-cache</td>
</tr>
<tr>
<td>Write-back (WB)</td>
<td></td>
</tr>
</tbody>
</table>
Bad News I: Timing Anomalies

We show such a pipeline has timing anomalies:

Toward Compact Abstractions for Processor Pipelines
Maximal cost of an additional cache miss?

Intuitively: main memory latency

Unfortunately: ~ 2 times main-memory latency
  - ongoing instruction fetch may block load
  - ongoing load may block instruction fetch
Key Insight: Anomalies Require Non-Monotonicity

In the blue state, each instruction has the same or more progress than in the red state.

The program has terminated.

The program has not yet terminated.

The program has terminated.
Monotonicity Enables Predictability 1/2

Theorem 1 (Timing Anomalies):
Monotonicity implies absence of timing anomalies.

\[ \text{local worst case} \leq \text{local best case} \leq \ldots \leq \text{by monotonicity} \]
Theorem 2 (Timing Compositionality): Monotonicity enables the derivation of sound penalties.
How to Achieve Monotonicity?

1. Pipelining is not the reason for non-monotonicity!
2. The shared resource memory may be accessed out-of-order!
Strictly In-Order Pipelines: Definition

**Definition (Strictly In-Order):**
We call a pipeline *strictly in-order* if each *resource* processes the instructions in program order.

- Enforce memory operations (instructions and data) in-order (common memory as resource)
- Block instruction fetch until no potential data accesses in the pipeline
Theorem 1 (Monotonicity):
In the strictly in-order pipeline progress of an instruction is monotone in the progress of other instructions.

Corollary (Timing Anomalies and Timing Compositionality):
In the strictly in-order pipeline
• does not have timing anomalies, and
• admits compositional analysis with natural penalties.
Performance:
Strictly in-order pipeline is about 6% slower than regular in-order pipeline.

→ Preserves most of the benefits of pipelining.

Predictability:
~4x faster single-core analysis
~32x faster multi-core analysis
Automating the Predictability Proofs

Formalization of Processor + Formalization of Property = SMT Solver

[Diagram showing a green check mark and a red X]
Processor states map dynamic instruction instances to their progress $\rightarrow$ **infinite state space**

Suffices to consider a finite window:

- are and stay in $\textit{post}$
- relevant instruction window
- might be in the pipeline
- could be fetched within $x$ cycles
- are and stay in $\textit{pre}$ during next $x$ cycles

---

**Challenges in Proof Automation**
Define formula that is **unsatisfiable**, if *transition relation* is **monotonic**.

\[ p_1 \sqsubseteq p_2 \land \text{cycle}(p_1, p'_1) \land \text{cycle}(p_2, p'_2) \land \neg p'_1 \sqsubseteq p'_2 \]

However, the formula is **satisfiable**!

Need to capture reachable states...
Proof of Monotonicity: Second Attempt

Define formula that is **unsatisfiable**, if *transition relation* is **monotonic**.

\[
\text{validPipelineState}(p_1) \land \text{validPipelineState}(p_2) \land \\
p_1 \sqsubseteq p_2 \land \text{cycle}(p_1, p'_1) \land \text{cycle}(p_2, p'_2) \land \neg p'_1 \sqsubseteq p'_2
\]

This formula is indeed **unsatisfiable**!
See paper:
Sebastian Hahn, Jan Reineke:
Design and analysis of SIC: a provably timing-predictable pipelined processor core. Real-Time Systems, November 2019
Efficiency of SMT Proofs

We implemented the construction of the SMT formulas in Python using the worst-case situation that motivates where be satisfiable or unsatisfiable. In particular, the central monotonicity property of previous sections. Formulas that reason about a very limited execution window, and Python bindings of Z3. To speed up the construction of the formulas we used automatically.

Indeed, Z3 is able to verify the unsatisfiability of the above formula for a range of memory latencies from 2 to 12 cycles. As an example, the formula to prove the timing compositionality of an instruction cache miss is:

\[
\text{denotes the instruction that might trigger the accident, } i = [i_1, \ldots, i_k] = [m, 0, \ldots, 0], \quad i_k = m - 1, \quad i^\text{hit}_k = m - 1, \quad s, s^\text{hit} = 0, \quad v = \sum_{i \in W} \text{cycle}(i) \cdot s_i + \text{cycle}(i^\text{hit}) \cdot s_i^\text{hit}.
\]

Figure 12 shows the runtime of Z3 for different verification goals.

The runtime for the compositionality proofs, however, depends on the size of memory latencies from 2 to 12 cycles. The above formula can also be used in a different verification goals.

The runtime in hours takes only seven seconds to verify.

\[
\text{proof } \text{monotonicity of SIC} \quad 7s
\]

\[
\text{proof } \text{non-monotonicity of textbook in-order} \quad < 1s
\]

\[
\text{proof } \text{anomaly-freedom w.r.t. cache} \quad < 1s
\]

\[
\text{proof } \text{compositionality w.r.t. interrupt} \quad < 1s
\]

\[
\text{proof } \text{non-monotonicity of textbook in-order} \quad < 1s
\]

\[
\text{proof } \text{anomaly-freedom w.r.t. cache} \quad < 1s
\]

\[
\text{proof } \text{compositionality w.r.t. interrupt} \quad < 1s
\]

Compositionality w.r.t. instruction-cache and interrupts:

![Runtime vs. Memory Latency](image)
Conclusions and Future Work

*Key Insight:* Monotonicity enables Timing Predictability

- Strictly in-order pipeline is monotonic
- Predictability proofs can be automated
  - Translation of model to SMT still manual
  - Need to capture relevant invariants manually
  - Can we automate the process further?
Sebastian Hahn, Jan Reineke: Design and analysis of SIC: a provably timing-predictable pipelined processor core. RTSS 2018 (best student paper award)

Sebastian Hahn, Jan Reineke: Design and analysis of SIC: a provably timing-predictable pipelined processor core. Real-Time Systems, November 2019