SIC: Provably Timing-Predictable Strictly In-Order Pipelined Processor Core

Sebastian Hahn and Jan Reineke

RTSS, Nashville
December, 2018
SIC: Provably Timing-Predictable Strictly In-Order Pipelined Processor Core

Sebastian Hahn and Jan Reineke

RTSS, Nashville
December, 2018
State-of-the-Art Timing Analysis

Taskset running on Hardware platform

✓ All tasks meet their deadline.
× At least one task misses its deadline.
State-of-the-Art Timing Analysis

Taskset running on Hardware platform

- Task-Level Analysis

- All tasks meet their deadline.
- At least one task misses its deadline.

\[ R_i = C_i + \sum_{j \in hp(i)} \lceil R_i T_j \rceil \cdot C_j + I_{bus}(i, R_i) \cdot \text{penalty} + \ldots \]

\[ I_{bus}(i, R_i) : \text{worst-case interference on shared bus} \]

Requires Compositionality for Soundness

Does not hold even for simple hardware platforms [Hahn et al., RTNS’16]
State-of-the-Art Timing Analysis

Taskset running on Hardware platform per task

Task-Level Analysis

characteristics per task, e.g. WCET $C_i$

✓ All tasks meet their deadline.

× At least one task misses its deadline.

Requires Compositionality for Soundness

Does not hold even for simple hardware platforms [Hahn et al., RTNS'16]

Sebastian Hahn

Strictly In-Order Core

RTSS 2018 2 / 17
State-of-the-Art Timing Analysis

Taskset running on Hardware platform

Task-Level Analysis

characteristics per task, e.g. WCET $C_i$

Schedulability Analysis

✓ All tasks meet their deadline.
× At least one task misses its deadline.

Requires Compositionality for Soundness

Does not hold even for simple hardware platforms [Hahn et al., RTNS'16]
Inside Schedulability Analysis:

\[ R_i := C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \]
State-of-the-Art Timing Analysis

Multi-Core System

Taskset running on Hardware platform

per task

Task-Level Analysis

characteristics per task, e.g.

Schedulability Analysis

✓ All tasks meet their deadline.

× At least one task misses its deadline.

Inside Schedulability Analysis:

\[ R_i := C_i + \sum_{j \in hp(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor \cdot C_j \]
State-of-the-Art Timing Analysis

Multi-Core System

![Diagram]

Taskset running on Hardware platform

Task-Level Analysis

per task

characteristics per task, e.g. WCET $C_i$, #BusAcc., ...

Schedulability Analysis

✓ All tasks meet their deadline.

× At least one task misses its deadline.

Inside Schedulability Analysis:

$$R_i := C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j + I_{bus}^i(i, R_i) \cdot \text{penalty} + \ldots$$

$I_{bus}^i(i, R_i)$: worst-case interference on shared bus
State-of-the-Art Timing Analysis

Multi-Core System

Taskset running on Hardware platform

Task-Level Analysis

per task

characteristics per task, e.g. WCET $C_i$, #BusAcc., ...

Schedulability Analysis

✓ All tasks meet their deadline.

✗ At least one task misses its deadline.

Inside Schedulability Analysis:

$$R_i := C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j + l_{bus}(i, R_i) \cdot \text{penalty} + \ldots$$

Requires **Compositionality** for Soundness

- Does not hold even for simple hardware platforms [Hahn et al., RTNS’16]
Why Task-Level Analysis is Expensive

Microarchitectural State Space Exploration

- Set of System States
- Processor Cycle

- Cache hit
- Cache miss
- Timing Anomalies
- Longest Path Search
- Processor Cycle
- System States
- Microarchitectural State Space Exploration
- Task-Level Analysis
- Expensive
Why Task-Level Analysis is Expensive

Microarchitectural State Space Exploration
+ Longest Path Search

Set of System States
→ Processor Cycle
Why Task-Level Analysis is Expensive

Microarchitectural State Space Exploration + Longest Path Search

Set of System States

Processor Cycle

cache hit

cache miss

Sebastian Hahn
Strictly In-Order Core
Why Task-Level Analysis is Expensive

Microarchitectural State Space Exploration + Longest Path Search

State Space Explosion due to **Timing Anomalies**
Compositionality + Anomaly Freedom = Timing Predictability
Timing Anomalies and Execution Progress

- Cache miss
- Cache hit

- Cycle behaviour is non-monotonic
- Cycle behaviour is monotonic

⇒ No timing anomalies
Timing Anomalies and Execution Progress

- Cache miss
- Cache hit

Cycle behaviour is non-monotonic $\Rightarrow$ No timing anomalies

Cycle behaviour is monotonic
Timing Anomalies and Execution Progress

Cache miss → less progress

Cycle behaviour is non-monotonic

Cache hit → more progress

Cycle behaviour is monotonic

⇒ No timing anomalies
Timing Anomalies and Execution Progress

- Cache miss
- Less progress
- Cycle behaviour is non-monotonic
- No timing anomalies

- Cache hit
- More progress
Timing Anomalies and Execution Progress

Timing anomaly $\Rightarrow$ Cycle behaviour is non-monotonic
Timing Anomalies and Execution Progress

Timing anomaly $\Rightarrow$ Cycle behaviour is non-monotonic

Cycle behaviour is monotonic $\Rightarrow$ No timing anomalies
Monotonicity w.r.t. Progress is Key to ...

Anomaly Freedom
Monotonicity w.r.t. Progress is Key to ...  
Anomaly Freedom
Monotonicity w.r.t. Progress is Key to ... Anomaly Freedom

\[ \text{finish}(c_{\text{miss}}) \geq \text{finish}(c_{\text{hit}}) \]
Monotonicity w.r.t. Progress is Key to ...

Timing Compositionality with Penalty $p$

$$c_{hit} = c_{miss} + p \text{ cycles}$$

$C_{rectified}$
Monotonicity w.r.t. Progress is Key to ...

Timing Compositionality with Penalty $p$

$$ finish(c_{\text{miss}}) = finish(c_{\text{rectified}}) + p $$
Monotonicity w.r.t. Progress is Key to ...
Timing Compositionality with Penalty $p$

\[ \text{finish}(c_{\text{miss}}) = \text{finish}(c_{\text{rectified}}) + p \]
Monotonicity w.r.t. Progress is Key to ...

Timing Compositionality with Penalty $p$

\[ \text{finish}(c_{\text{miss}}) = \text{finish}(c_{\text{rectified}}) + p \]
\[ \text{finish}(c_{\text{rectified}}) \leq \text{finish}(c_{\text{hit}}) \]
Monotonicity w.r.t. Progress is Key to ...

Timing Compositionality with Penalty $p$

\[ \text{finish}(c_{\text{miss}}) = \text{finish}(c_{\text{rectified}}) + p \]

\[ \text{finish}(c_{\text{rectified}}) + p \leq \text{finish}(c_{\text{hit}}) + p \]
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

<table>
<thead>
<tr>
<th>Fetch</th>
<th>add, 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>load ready</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Write-Back</td>
<td></td>
</tr>
</tbody>
</table>

Relative progress of instructions influences order of bus accesses.

Sebastian Hahn

Strictly In-Order Core

RTSS 2018
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

![Diagram showing relative progress of instructions influencing order of bus accesses.](image-url)
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
<th>Memory</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add, 5</td>
<td>load ready</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cycle</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
\text{add, 5} \\
\text{load ready} \\
\end{array}
\]

\[
\begin{array}{c}
\text{add} \\
\text{load ready} \\
\end{array}
\]

\[
\begin{array}{c}
\text{cycle} \\
\text{add fetch miss} \\
\text{load data miss} \\
\end{array}
\]
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

Relative progress of instructions influences order of bus accesses

Sebastian Hahn
Strictly In-Order Core
RTSS 2018 9 / 17
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

Fetch
Decode
Execute
Memory
Write-Back

$\text{add, 5}$
$\text{load ready}$

$\text{add}$
$\text{load ready}$

$\text{add, 4}$
$\text{load ready}$

$\text{add}$
$\text{load, ml}$

$\text{cycle}$

$\text{add fetch miss}$
$\text{load data miss}$

Sebastian Hahn
Strictly In-Order Core
RTSS 2018 9 / 17
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

Fetch
Decode
Execute
Memory
Write-Back

add, 5
load ready

\[
\begin{align*}
\text{add} & \quad 5 \\
\text{load ready} & \\
\end{align*}
\]

\[
\begin{align*}
\text{add} & \\
\text{load ready} & \\
\end{align*}
\]

cycle

\[
\begin{align*}
\text{add, 4} & \\
\text{load ready} & \\
\end{align*}
\]

\[
\begin{align*}
\text{add} & \\
\text{load, } ml & \\
\end{align*}
\]

add fetch miss
load data miss

Sebastian Hahn
Strictly In-Order Core
RTSS 2018
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

Relative progress of instructions influences order of bus accesses
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

Relative progress of instructions influences order of bus accesses
Relative progress of instructions influences order of bus accesses

Idea: Make order of bus accesses independent of relative progress
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline

Relative progress of instructions influences order of bus accesses

Idea: Make order of bus accesses independent of relative progress

Mechanism: Prioritise data accesses over instruction fetches
SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline
Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline
SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline
SIC: Strictly In-Order Pipelined Core
Enforce Monotonic Behaviour

Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline

Diagram:
- Fetch
- Decode
- Execute
- Memory
- Write-Back
- Instruction Cache
- Data Cache
- Memory

Fetch ins
ins misses

load/store pending? ⇒ strict bus access order
Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline
SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline

- Fetch
- Decode
- Execute
- Memory
- Write-Back

Instruction Cache

Data Cache

Memory

load/store pending?

fetch ins

ins misses

blocked

⇒ strict bus access order
Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline.

⇒ strict bus access order
Design and Analysis of SIC: A Provably Timing-Predictable Pipelined Processor Core

Sebastian Hahn and Jan Reineke
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
{sebastian.hahn, reinke}@cs.uni-saarland.de

Abstract—We introduce the strictly in-order core (SIC), a timing-predictable pipelined processor core. All in-order pipelines admit the property of timing predictability, which ensures precise and efficient worst-case execution time (WCET) and multi-core timing analysis.

SIC's key underlying property is the monotonicity of its transition relation w.r.t. a natural partial order on its microarchitectural states. This monotonicity is achieved by carefully eliminating some of the dependencies between consecutive instructions from a standard in-order pipeline design.

SIC preserves most of the benefits of pipelining: it is only about 6.7% slower than a conventional pipelined processor. Its timing predictability enables orders-of-magnitude faster WCET and multi-core timing analysis than conventional designs.

I. INTRODUCTION

One of the main challenges for timing analysis is the dependence of the execution time on the state of the underlying hardware platform. Even simple single-core processors feature stateful performance-enhancing mechanisms such as pipelines and caches. In the presence of such stateful mechanisms, an individual instruction's execution time may vary widely depending on the state of the hardware when the instruction is executed. For example, a cache miss usually takes significantly longer than a cache hit.

Simply assuming the worst-case latency of instructions throughout a program's execution would result in a dramatic overestimation of its worst-case execution time (WCET). To achieve accurate results, WCET analysis thus needs to precisely take into account in which hardware state a program's instructions are executed. State-of-the-art static WCET analysis tools [1] explore a program's possible executions on a given hardware platform by a combination of explicit and implicit techniques. It is desirable to employ implicit techniques, such as abstractions [2], to efficiently explore large sets of states. While precise and efficient abstractions are known for caches [3], the efficient implicit analysis of pipelining is impeded by the presence of timing anomalies [4], [5]. Due to timing anomalies it is not safe for WCET analysis to only explore "local worst-case" successors of pipeline states; nor is it possible to derive efficient abstractions in which sets of concrete pipeline states are represented by individual abstract pipeline states.

Timing analysis for applications deployed on multi-core processors is even more challenging. Multi-core processors share resources such as buses, caches, or memory channels among multiple cores. As a consequence, the execution time of a task depends on the interference on shared resources that it experiences due to co-running tasks on other cores.

As in single-core WCET analysis, assuming the worst-case latency upon every shared-resource access is not a viable option as it would result in high pessimistic execution time bounds. The greatest analysis precision would be achieved by fully-annotated timing analysis [6]-[8], such analyses simultaneously analyze the tasks running on different cores of a multi-core, precisely capturing all possible interleavings of resource accesses from different cores. Unfortunately, this approach appears to be practically infeasible for realistic systems due to the astronomical number of system states to explore. The most promising approach to multi-core timing analysis to date is compositional timing analysis [9]-[18], which can be seen as a natural extension of the classical two-step approach to timing analysis: low-level analysis, corresponding to classical WCET analysis, computes the "resource demand" of each task for each shared resource. Given such task characterizations, schedulability analysis then determines, whether each task can be guaranteed to meet its deadlines, accounting for the interference it may experience on each of the shared resources. Compositional timing analysis relies on the assumption that the response time of a task may be decomposed into contributions from different resources, which can then be efficiently analyzed separately.

We have shown in previous work that, unfortunately, even simple in-order pipelined cores feature timing anomalies and do not admit compositional timing analysis [19].

Based on preliminary ideas presented in [20], in this paper, we introduce the strictly in-order core (SIC), a pipelined processor core that is provably free of timing anomalies and that admits compositional timing analysis.

The starting point of this work has been the observation that the presence of timing anomalies in conventional in-order pipelines can be traced back to the non-monotonicity of their timing behavior. This non-monotonicity is due to dependencies between consecutive instructions, where progress of one instruction may be detrimental to the progress of another instruction. The key property of SIC is the monotonicity of its timing behavior, which is enforced by carefully eliminating some of the dependencies between instructions in the pipeline.

- Formal definition of progress
- Definition of strictly in-order behaviour
- Proof of monotonicity
- Corollary: anomaly freedom
- Corollary: timing compositionality
Evaluation Questions

1.) Performance Overhead of Enforcing Access Order

2.) Gain in Analysis Efficiency
Performance SIC versus Conventional In-Order

The diagram illustrates the performance degradation of SIC (Strictly In-Order Core) versus in-order execution across different cache sizes and memory latencies. The x-axis represents cache sizes ranging from 64 sets to 1024 sets, while the y-axis shows performance degradation from 0 to 1.5. The diagram indicates that as the cache size increases, the performance degradation remains constant, suggesting that SIC and conventional in-order cores perform similarly under these conditions. The memory latency is shown with values of 4 cycles, 12 cycles, and 100 cycles.
Performance SIC versus Conventional In-Order

FPGA implementation overhead 0.2%

The chart shows the performance degradation of SIC versus in-order FPGA simulation across different cache sizes and memory latencies. The performance degradation is normalized to the in-order performance, with values indicating the relative slowdown. The chart includes cache sizes of 64 sets, 256 sets, and 1024 sets, as well as memory latencies of 4 cycles, 12 cycles, and 100 cycles.
Performance SIC versus Conventional In-Order FPGA implementation overhead.

![Graph comparing performance degradation of SIC versus in-order FPGA simulation and static analysis.](chart)

- **Cache size:**
  - 64 sets
  - 256 sets
  - 1024 sets

- **Memory latency:**
  - 4 cycles
  - 12 cycles
  - 100 cycles
Performance Single PTARM-Thread versus SIC
A Precision-Timed (PRET) Microarchitecture Implementation [Liu et al., ICCD’12]

Performance degradation of PTARM versus SIC

Cache size: 64 sets, 256 sets, 1024 sets
Memory latency: 4 cycles, 12 cycles, 100 cycles

FPGA simulation vs static analysis
Task-Level Analysis Cost

slowdown in analysis runtime relative to SIC

cache size

memory latency

4 cycles

12 cycles

100 cycles

64 sets

256 sets

1024 sets

64 sets

256 sets

1024 sets

64 sets

256 sets

1024 sets

PTARM

Sebastian Hahn

Strictly In-Order Core

RTSS 2018 16 / 17
Task-Level Analysis Cost

![Graph showing slowdown in analysis runtime relative to SIC for different cache sizes and memory latencies. The x-axis represents cache sizes (64 sets, 256 sets, 1024 sets) and memory latencies (4 cycles, 12 cycles, 100 cycles). The y-axis represents slowdown in analysis runtime on a logarithmic scale. The bars indicate the slowdown for PTARM (blue) and in-order (anomalies) (red).]
Task-Level Analysis Cost

Compositional Base Bound [Hahn et al., RTNS’16]
Conclusions

Strictly in-order pipeline

- designed to behave monotonically
- provably timing-predictable: anomaly freedom + compositionality

yet performant

practical: low implementation overhead