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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

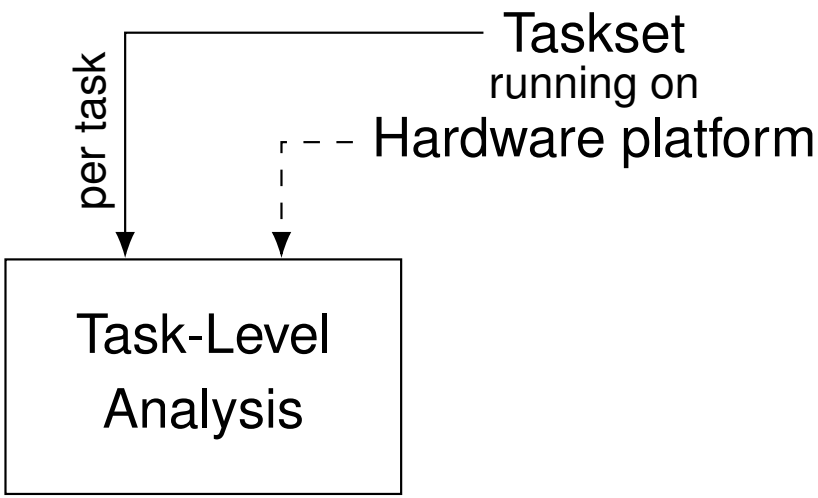
State-of-the-Art Timing Analysis

Taskset
running on
Hardware platform

✓ All tasks meet their deadline.

× At least one task misses its deadline.

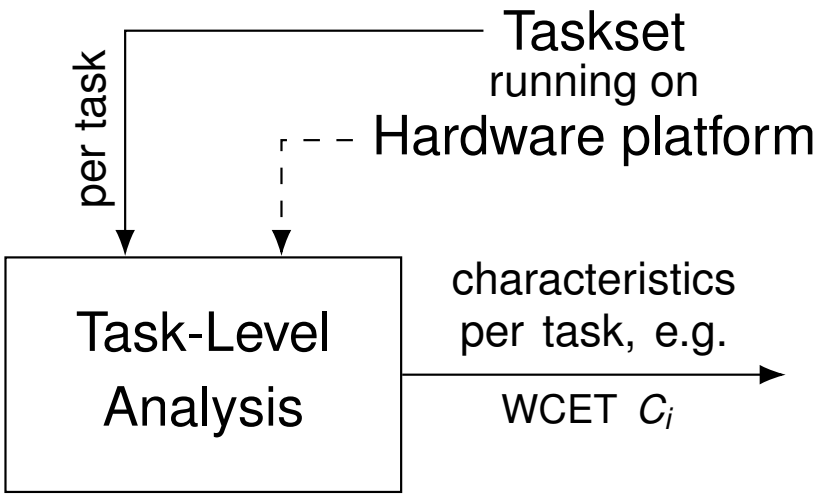
State-of-the-Art Timing Analysis



✓ All tasks meet their deadline.

✗ At least one task misses its deadline.

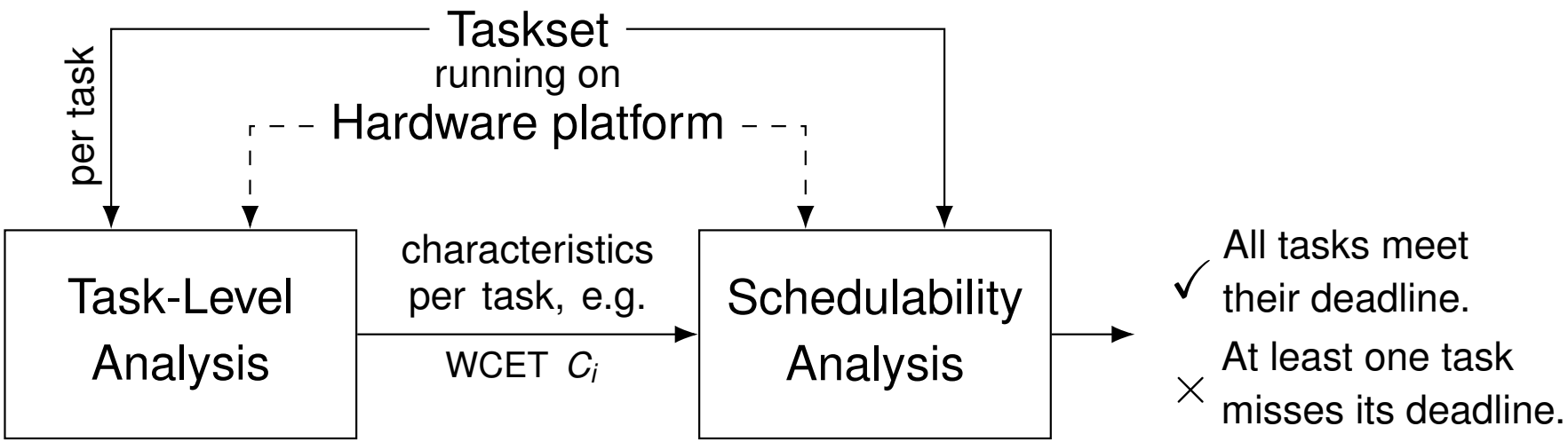
State-of-the-Art Timing Analysis



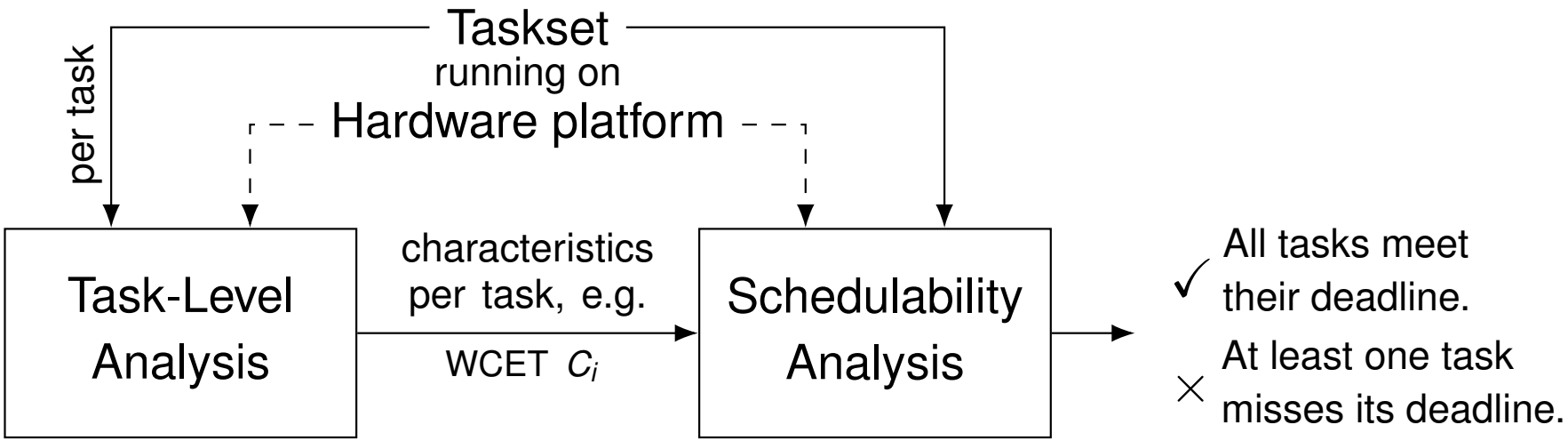
✓ All tasks meet their deadline.

✗ At least one task misses its deadline.

State-of-the-Art Timing Analysis



State-of-the-Art Timing Analysis

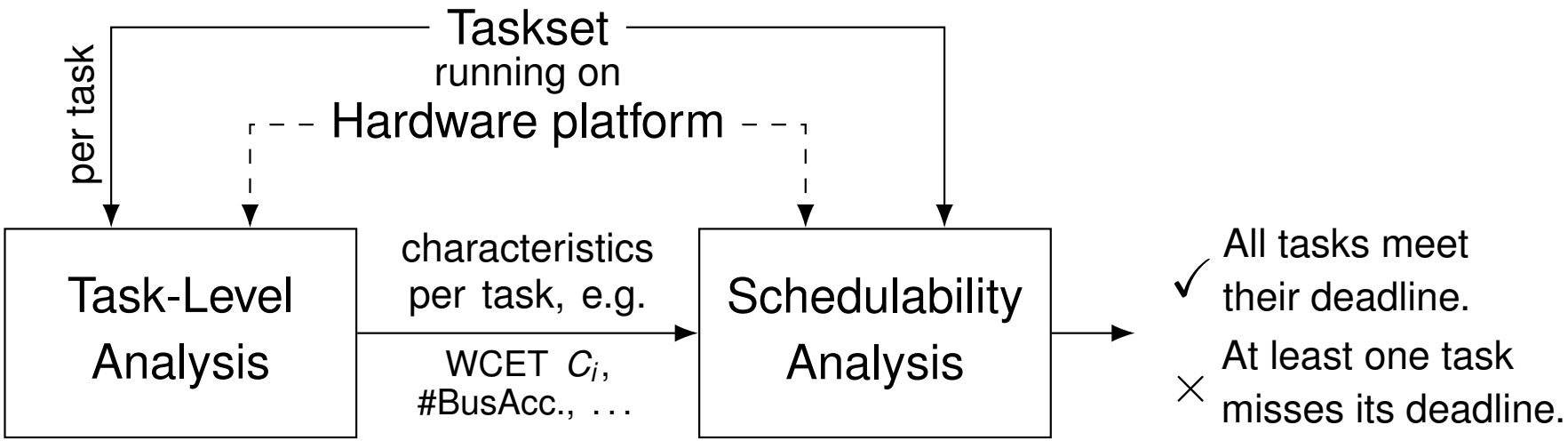


Inside Schedulability Analysis:

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

State-of-the-Art Timing Analysis

Multi-Core System

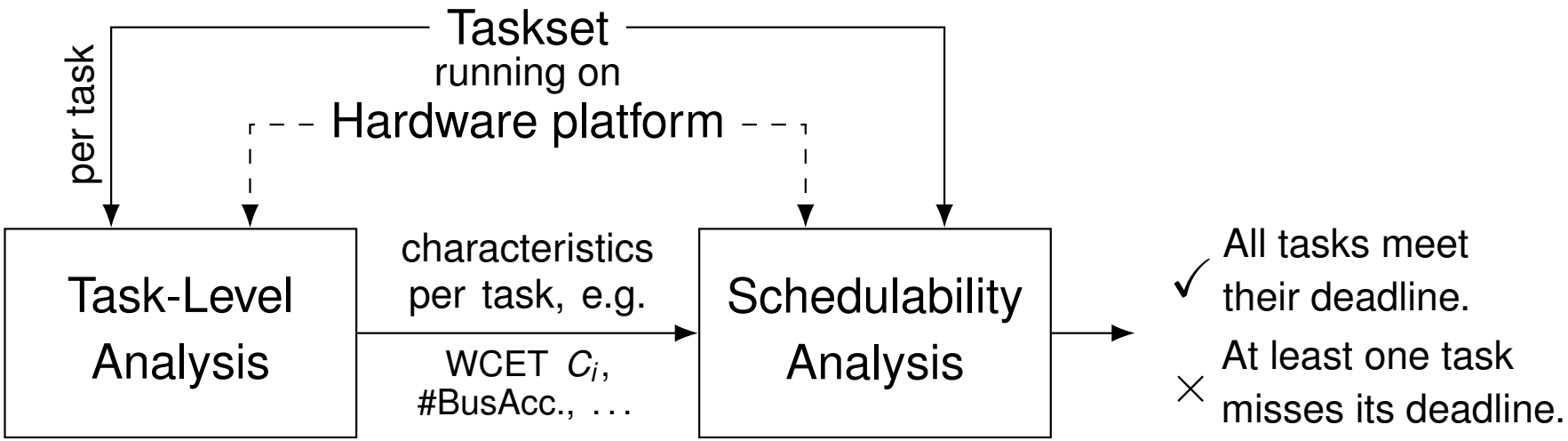


Inside Schedulability Analysis:

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

State-of-the-Art Timing Analysis

Multi-Core System



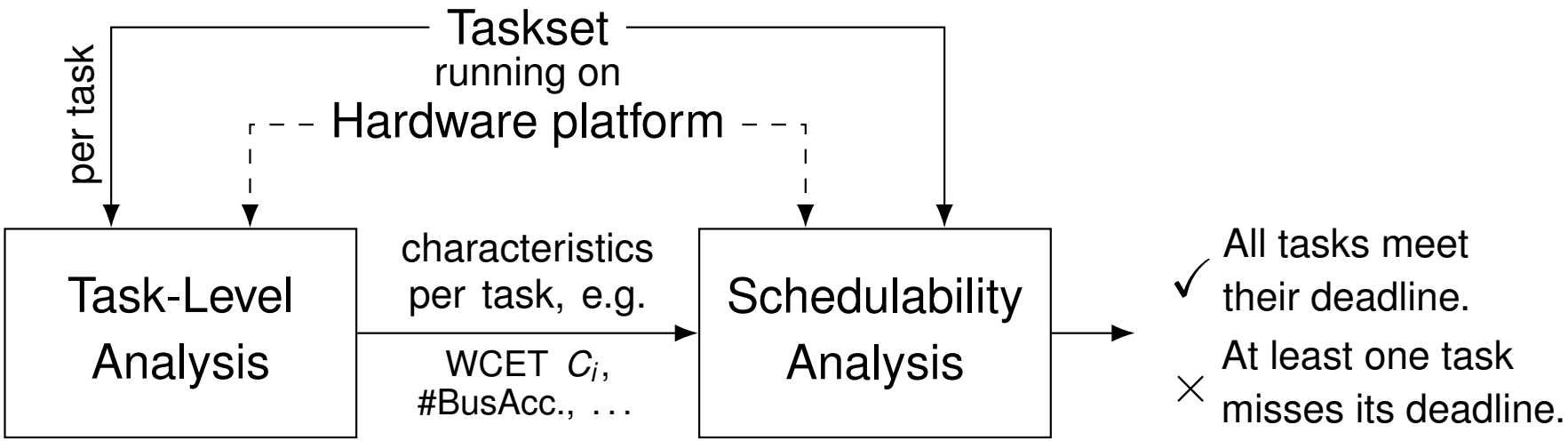
Inside Schedulability Analysis:

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

$I^{bus}(i, R_i)$: worst-case interference on shared bus

State-of-the-Art Timing Analysis

Multi-Core System



Inside Schedulability Analysis:

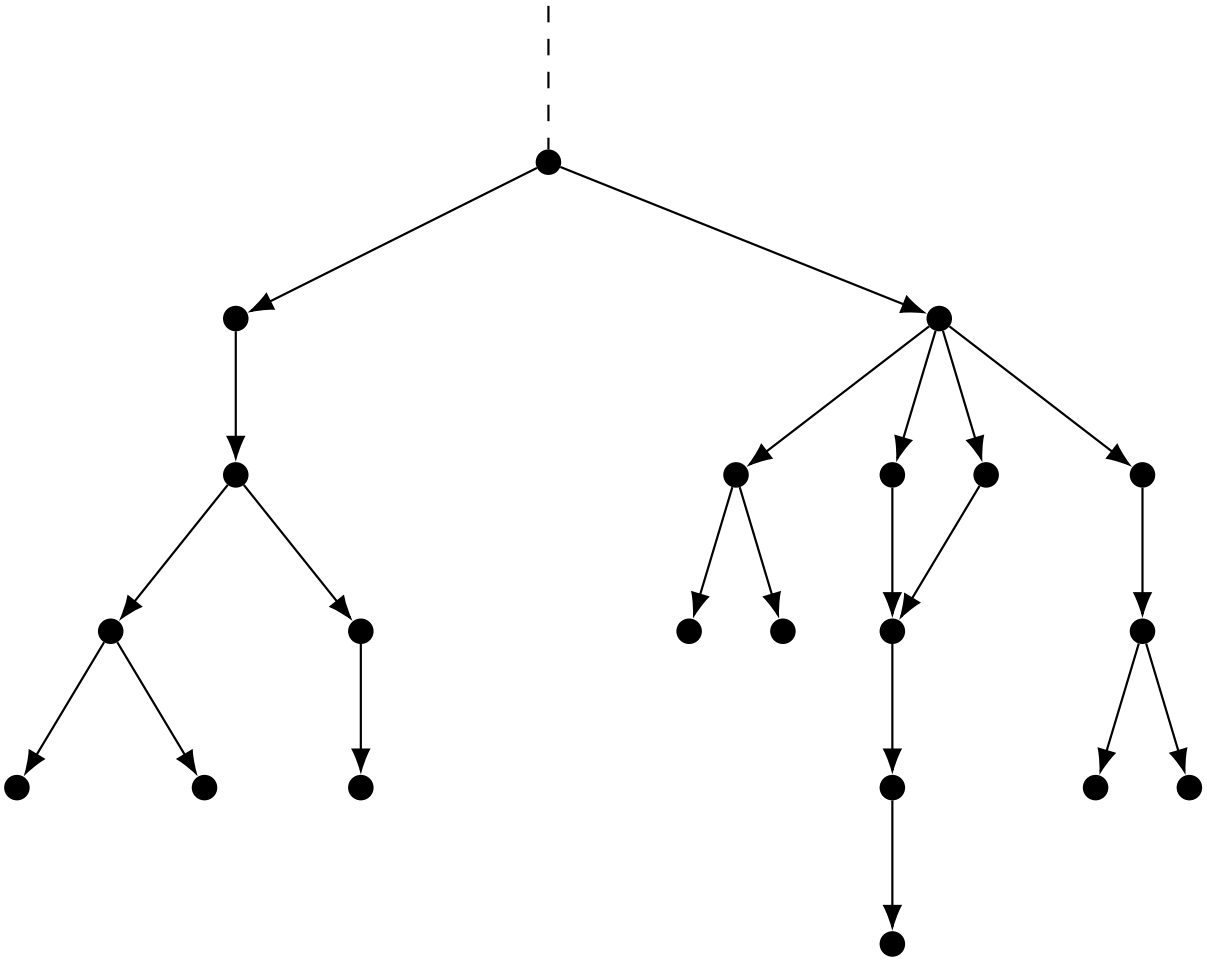
$$R_i := C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j + I^{bus}(i, R_i) \cdot penalty + \ldots$$

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



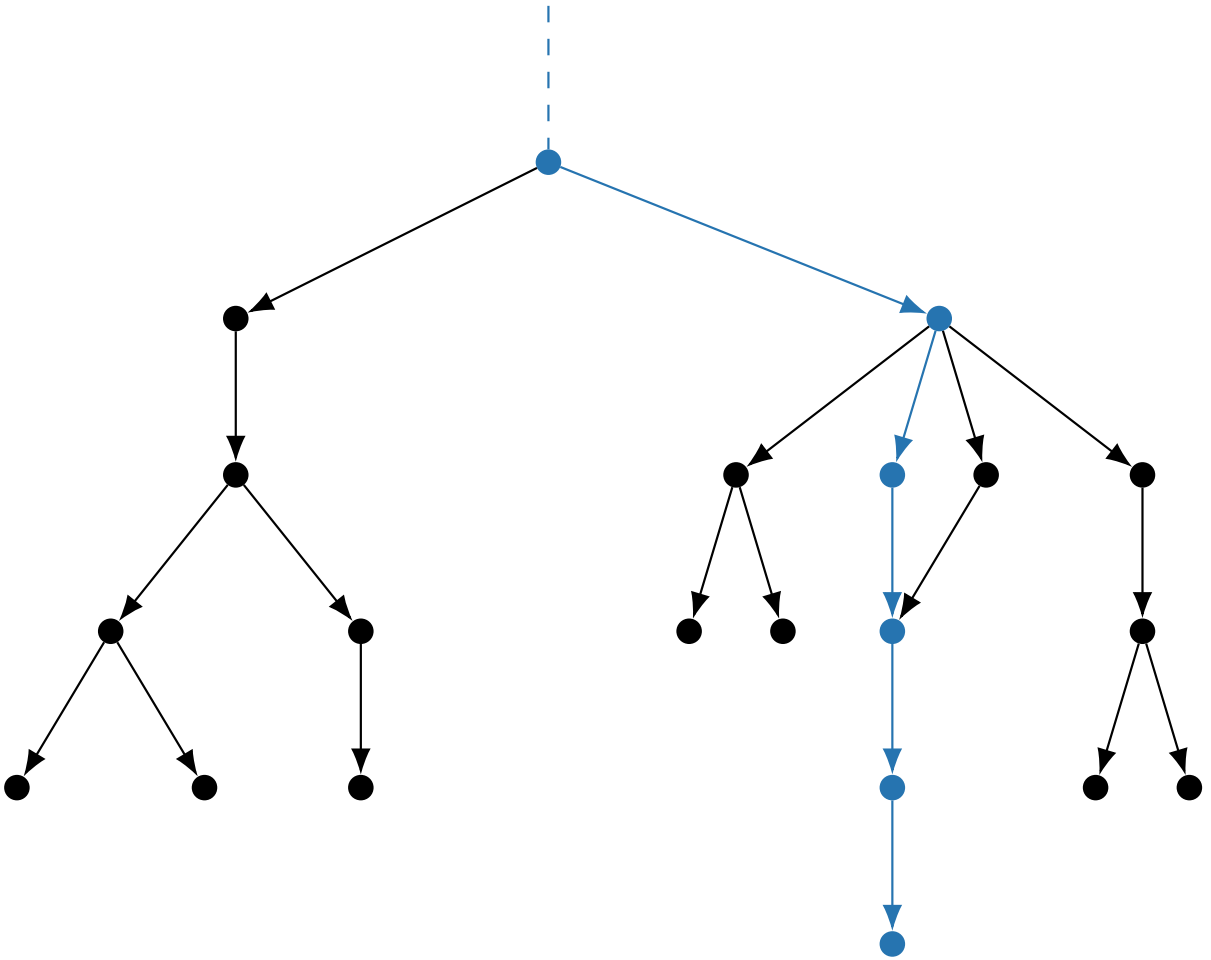
Set of System States

Processor Cycle

Why Task-Level Analysis is Expensive

Microarchitectural State Space Exploration

\+ Longest Path Search



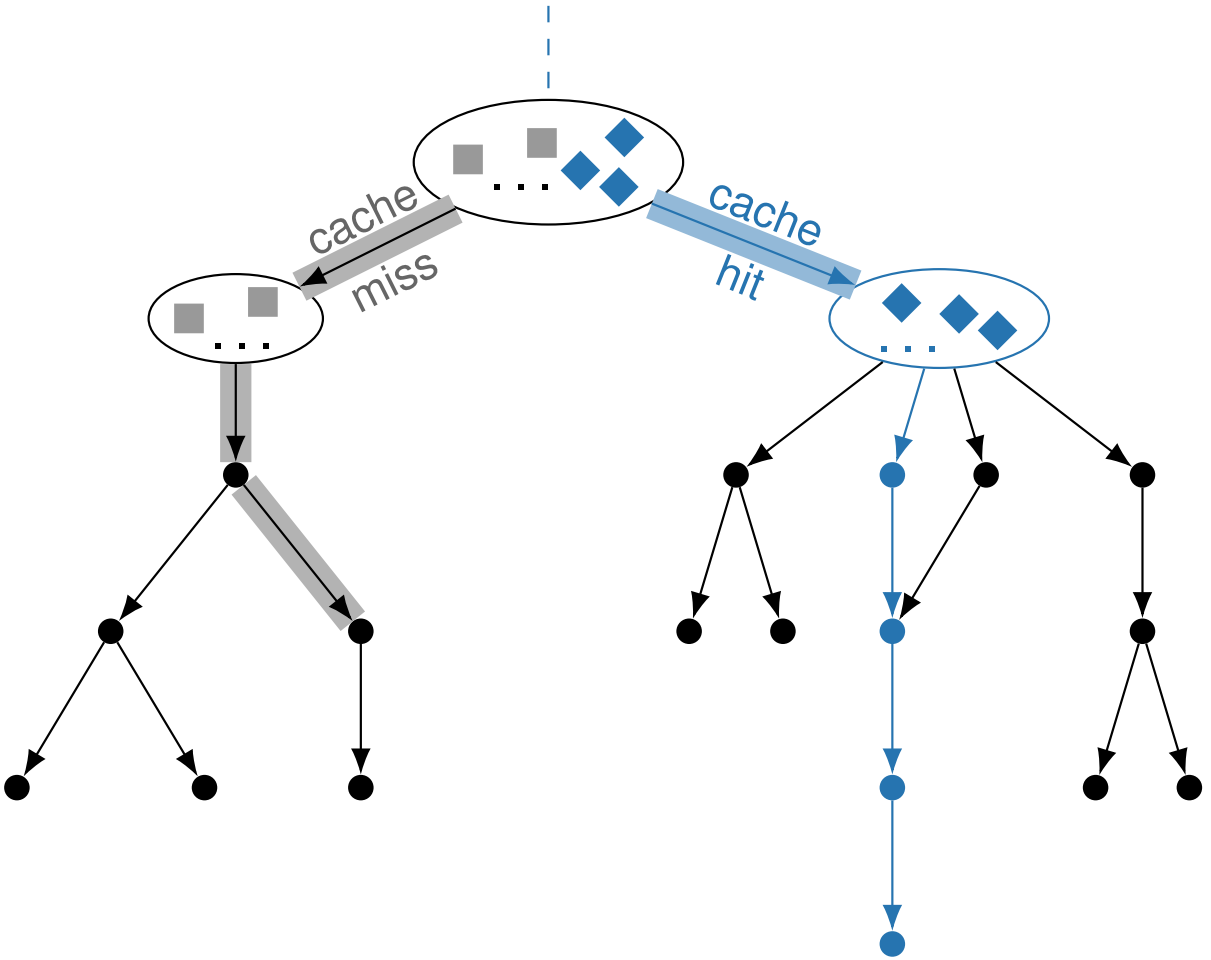
Set of System States

Processor Cycle

Why Task-Level Analysis is Expensive

Microarchitectural State Space Exploration

\+ Longest Path Search

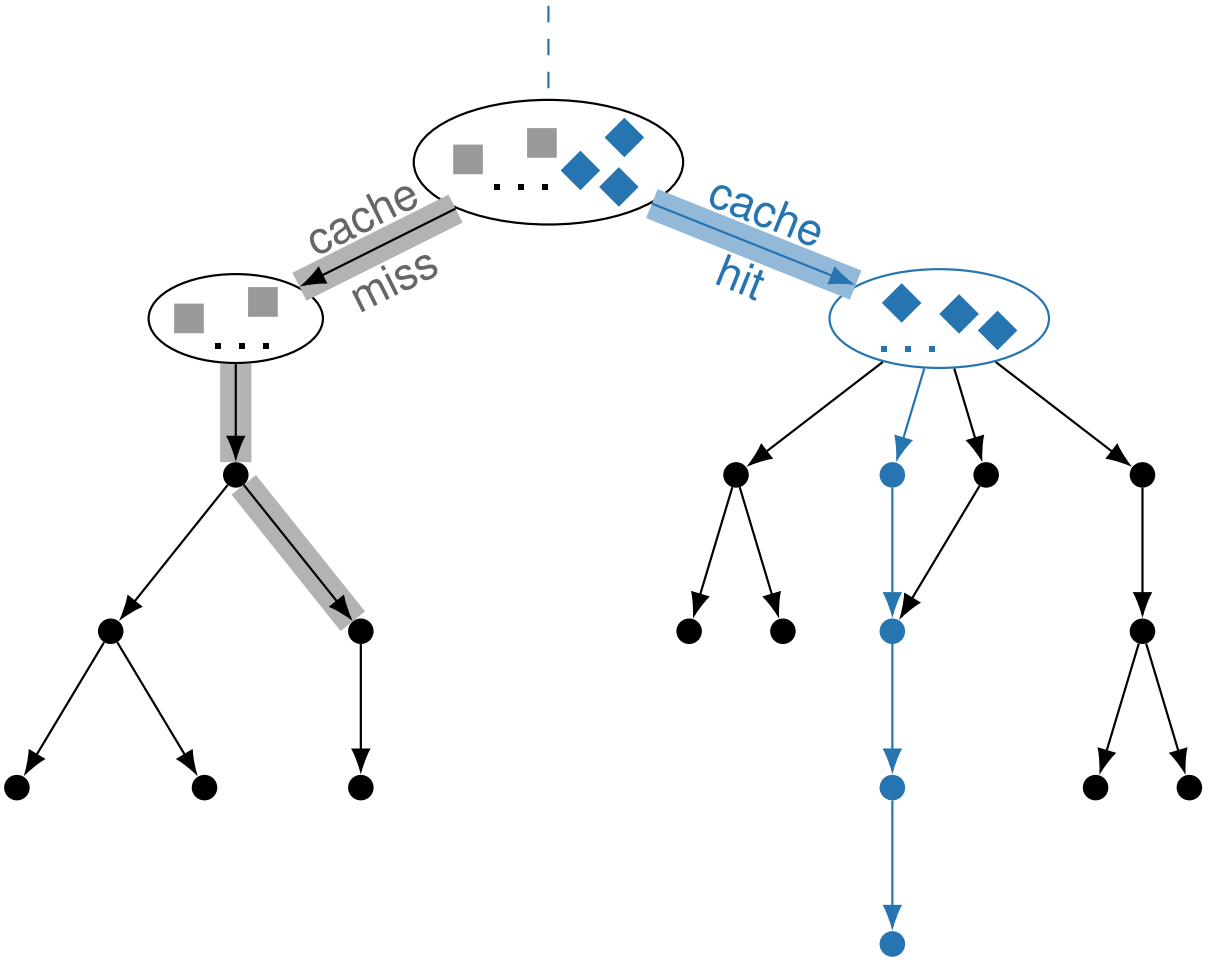


- Set of System States
- → Processor Cycle

Why Task-Level Analysis is Expensive

Microarchitectural State Space Exploration

\+ Longest Path Search



- Set of System States
- → Processor Cycle

State Space Explosion due to **Timing Anomalies**

Compositionality

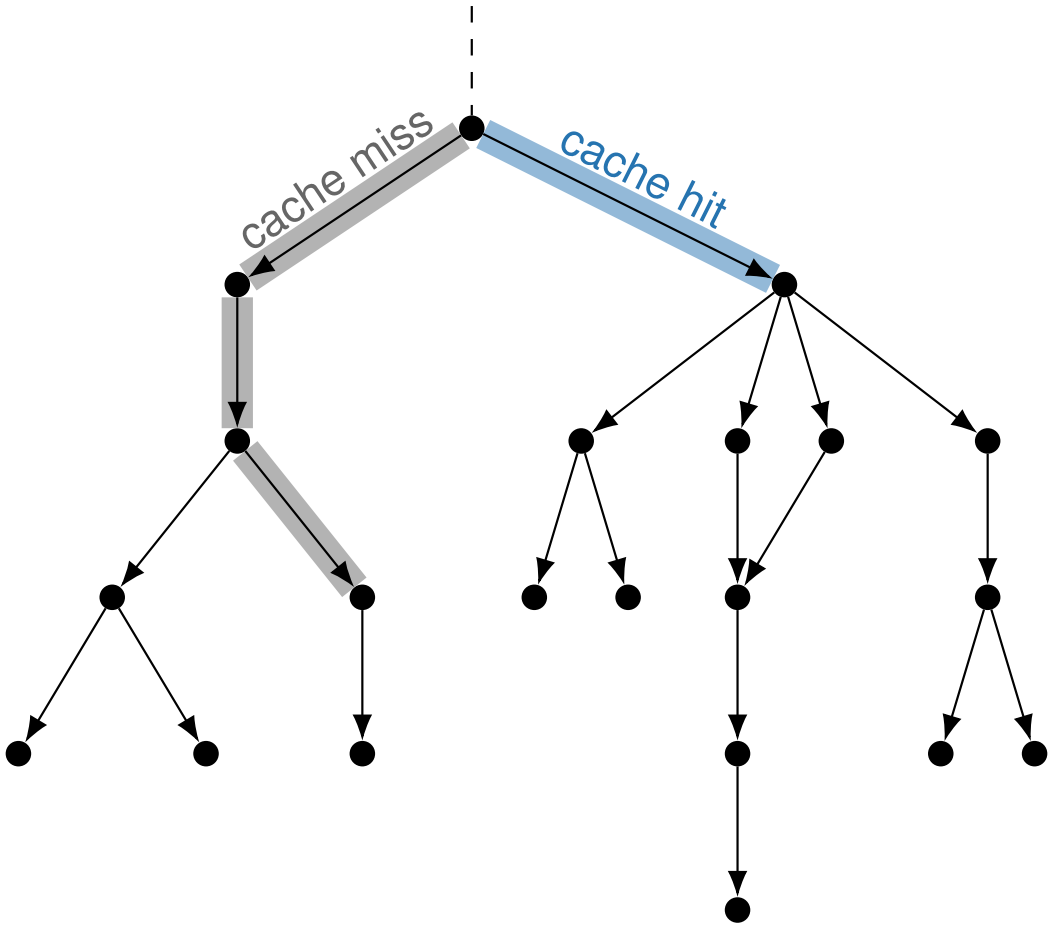
+

Anomaly Freedom

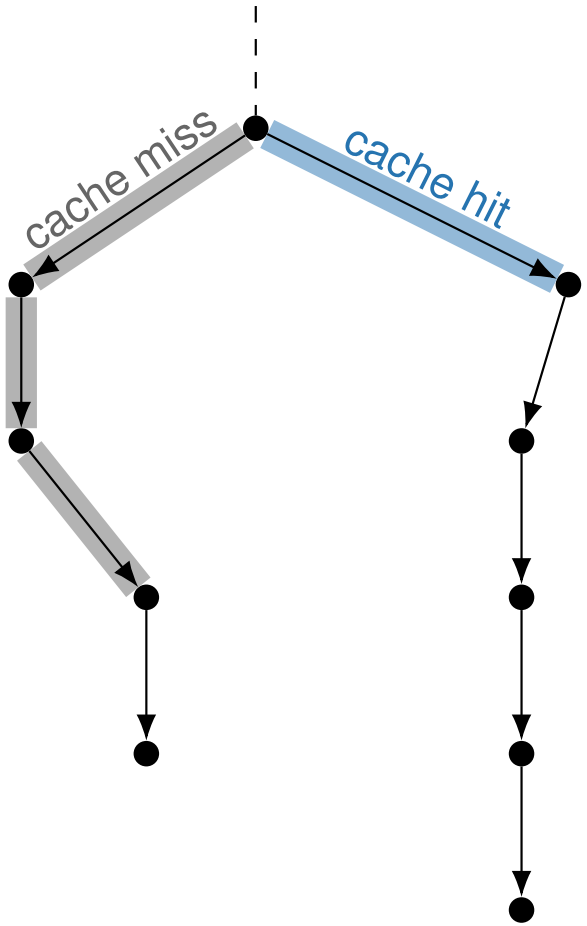
=

Timing Predictability

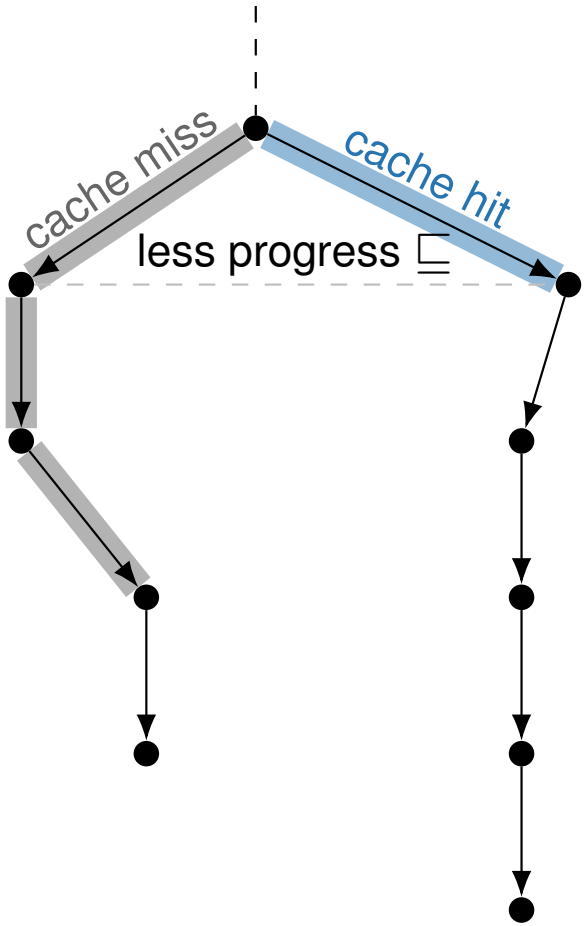
Timing Anomalies and Execution Progress



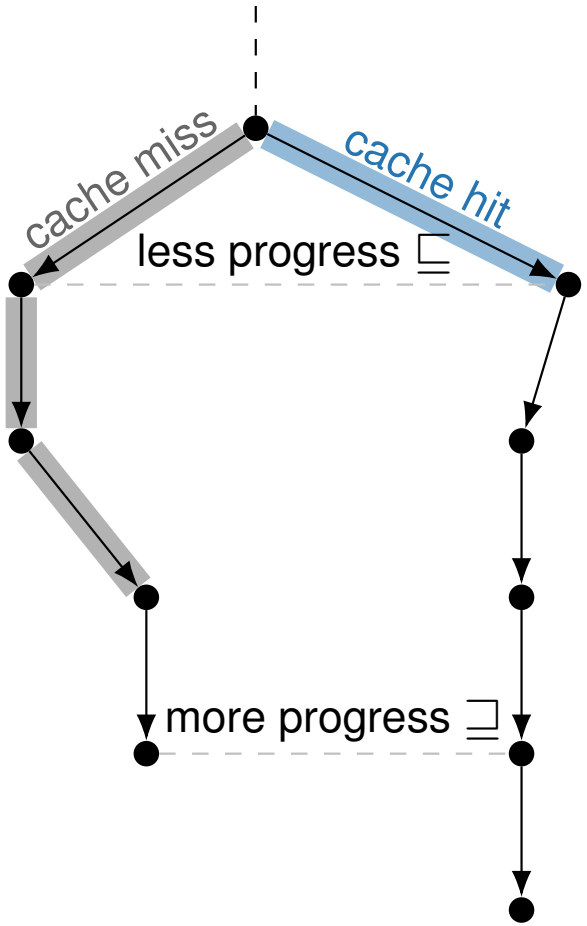
Timing Anomalies and Execution Progress



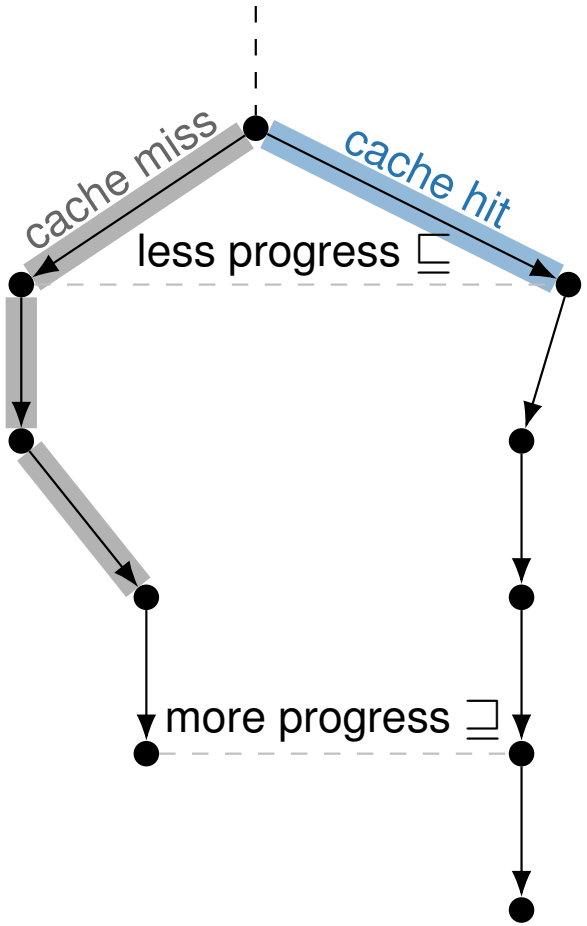
Timing Anomalies and Execution Progress



Timing Anomalies and Execution Progress

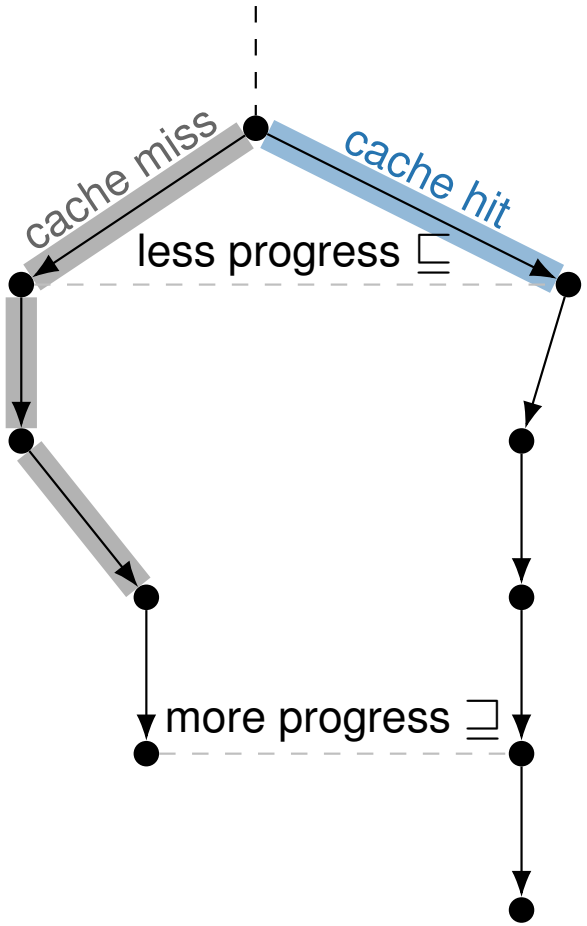


Timing Anomalies and Execution Progress



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

Timing Anomalies and Execution Progress

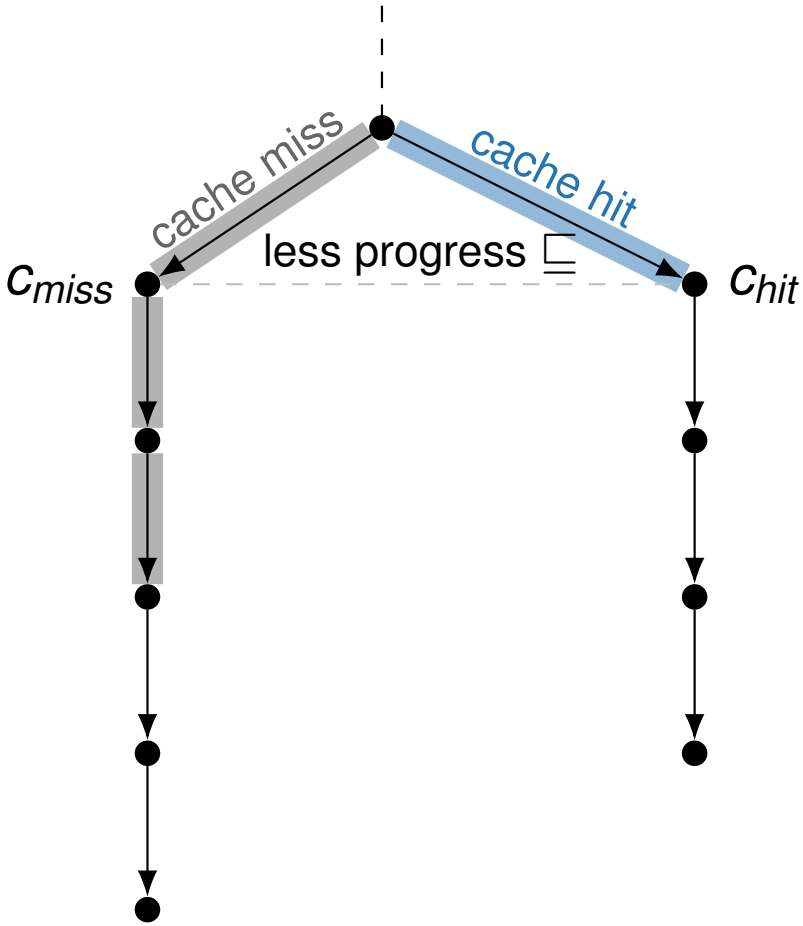


Timing anomaly $\Rightarrow$ Cycle behaviour is non-monotonic

Cycle behaviour is monotonic $\Rightarrow$ No timing anomalies

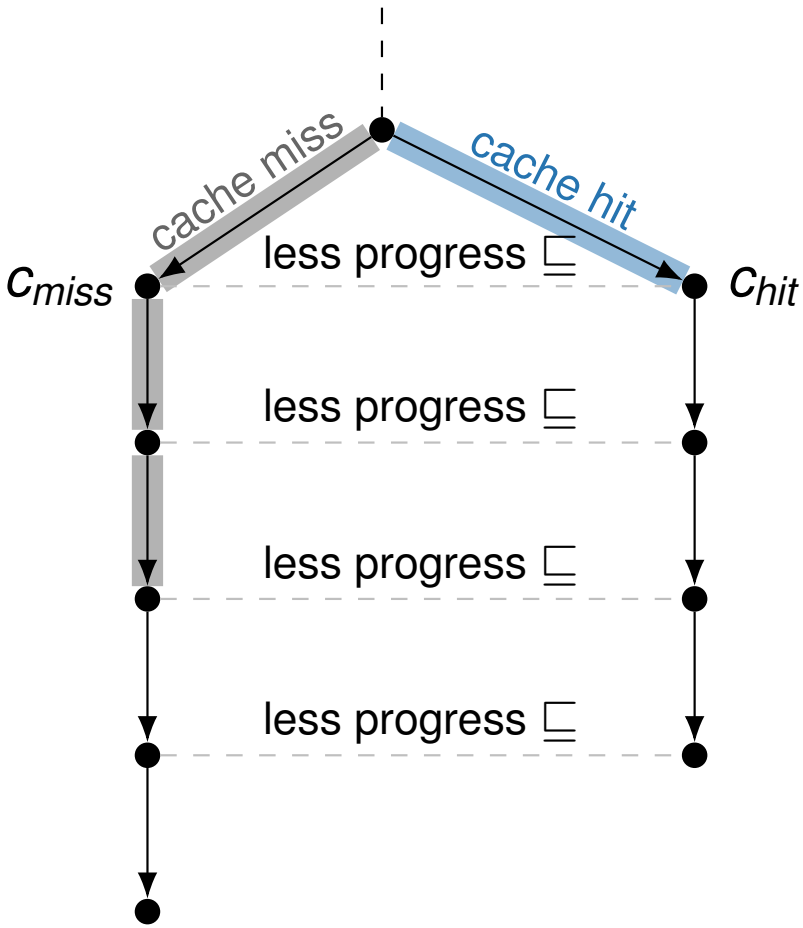
Monotonicity w.r.t. Progress is Key to ...

Anomaly Freedom



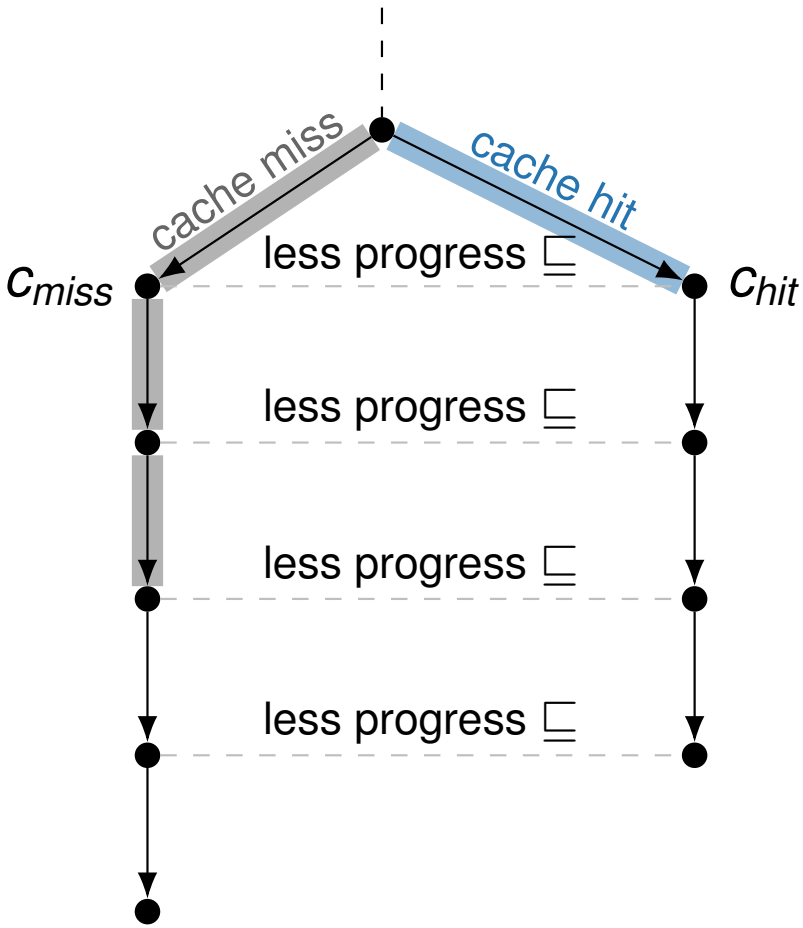
Monotonicity w.r.t. Progress is Key to ...

Anomaly Freedom



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

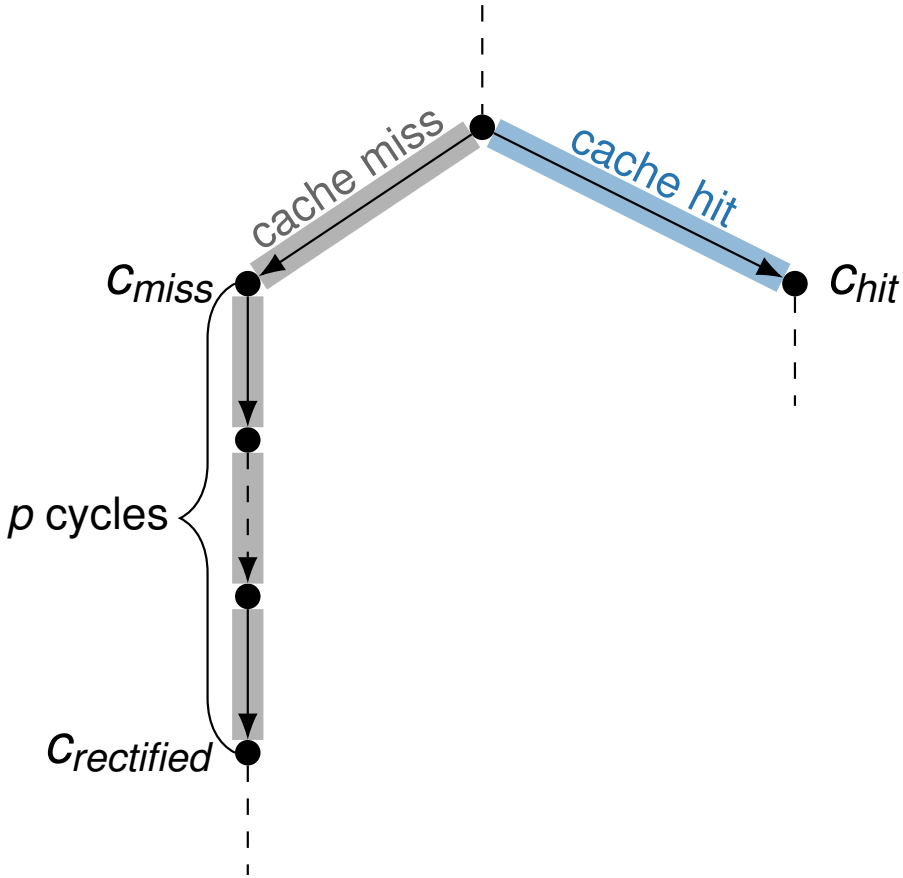
Anomaly Freedom



$$finish(c_{miss}) \geq finish(c_{hit})$$

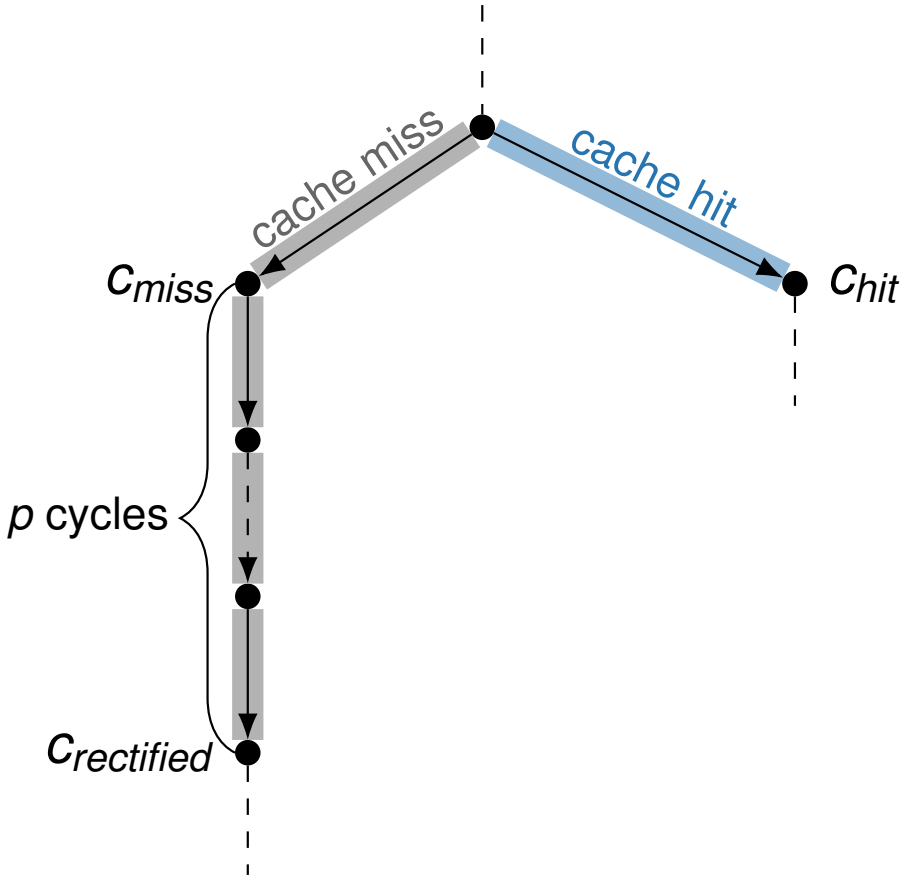
Monotonicity w.r.t. Progress is Key to ...

Timing Compositionality with Penalty *p*



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

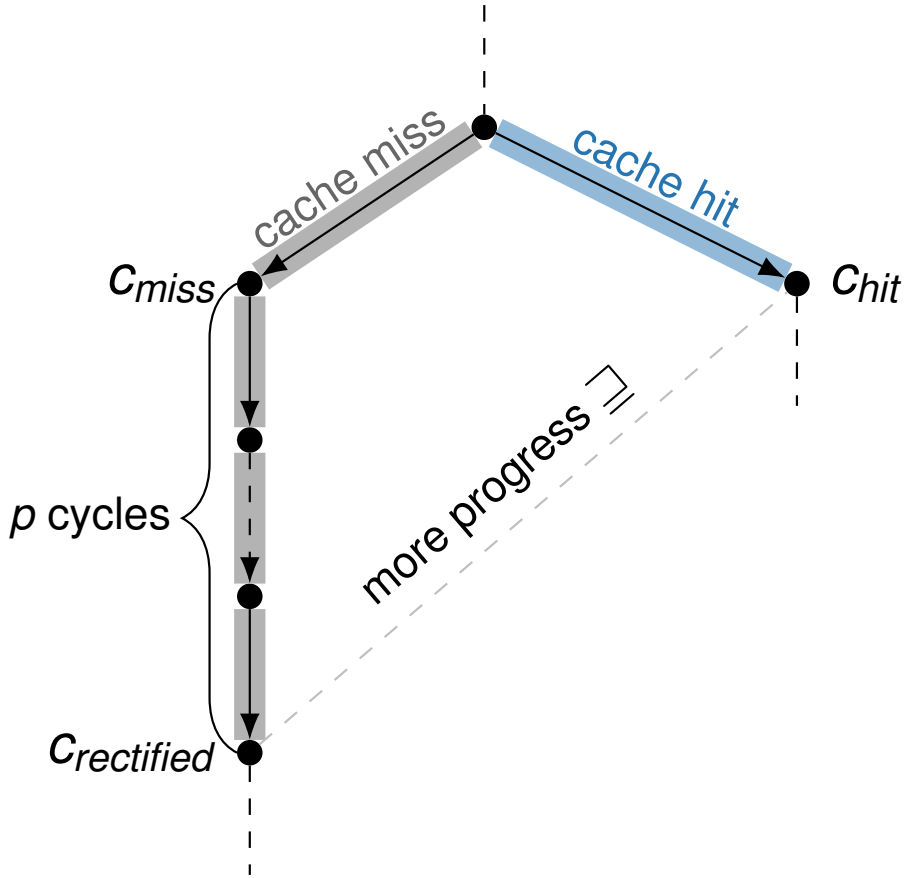
Timing Compositionality with Penalty p



$$finish(c_{miss}) = finish(c_{rectified}) + p$$

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

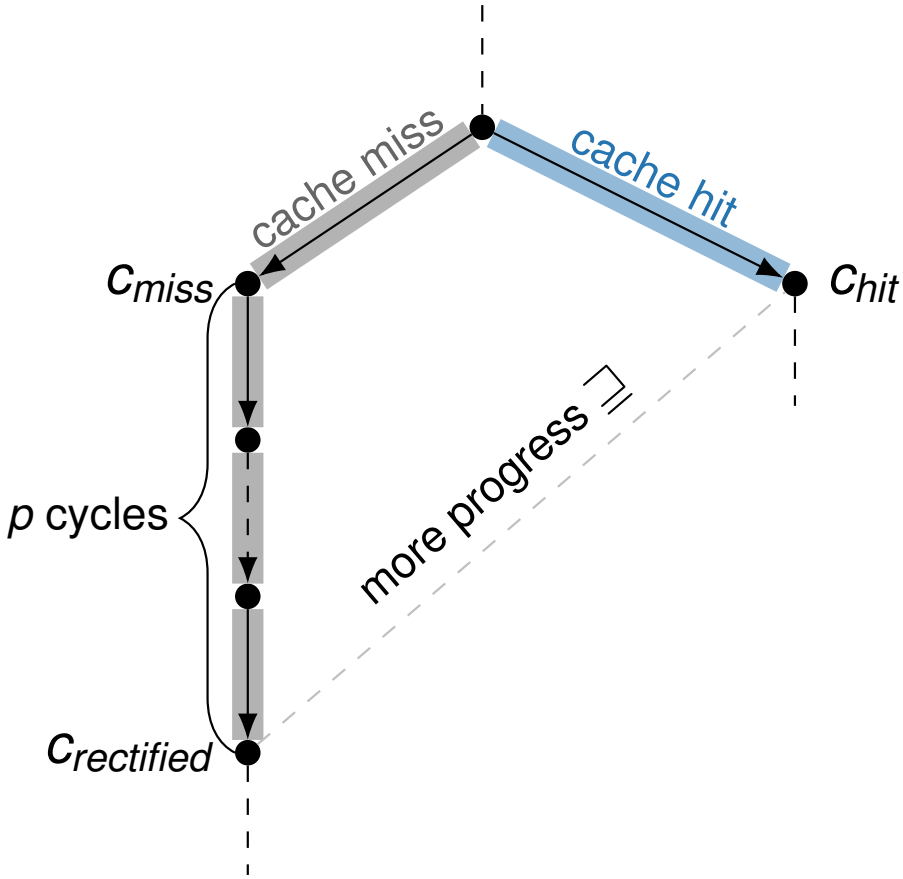
Timing Compositionality with Penalty p



$$finish(c_{miss}) = finish(c_{rectified}) + p$$

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

Timing Compositionality with Penalty p

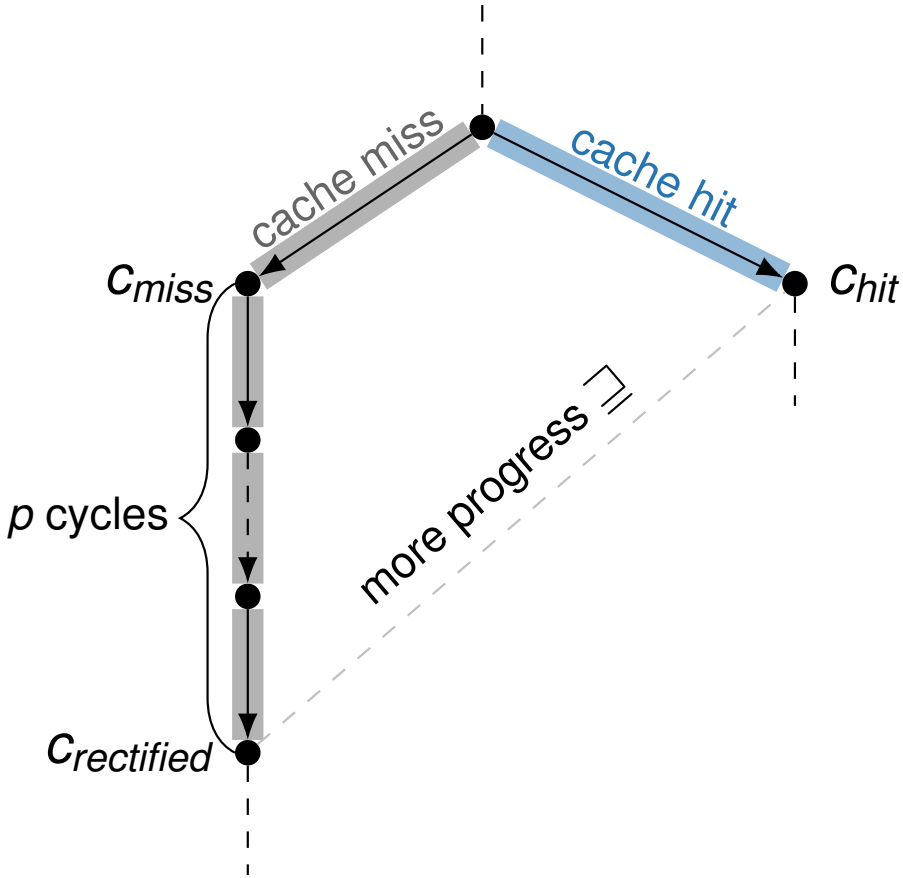


$$finish(c_{miss}) = finish(c_{rectified}) + p$$

$$finish(c_{rectified}) \leq finish(c_{hit})$$

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

Timing Compositionality with Penalty p

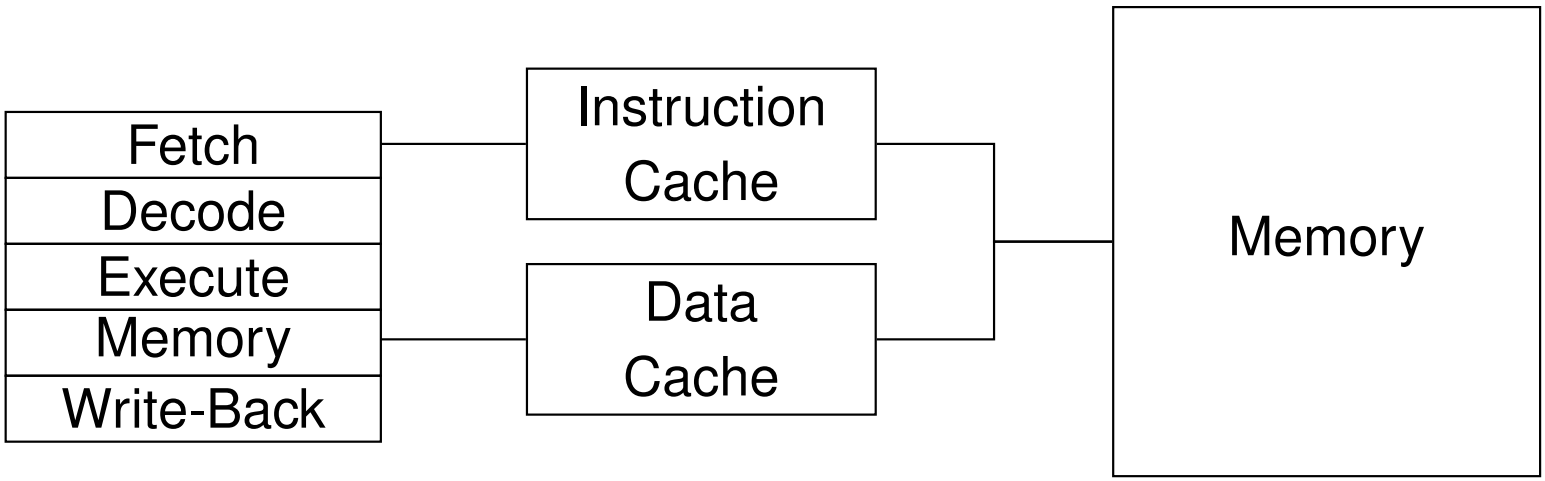


$$finish(c_{miss}) = finish(c_{rectified}) + p$$

$$finish(c_{rectified}) + p \leq finish(c_{hit}) + p$$

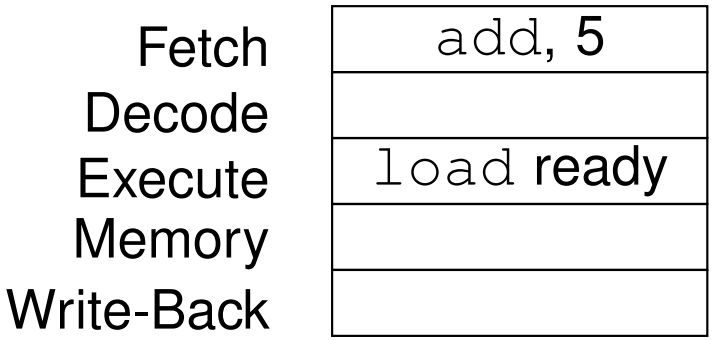
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline



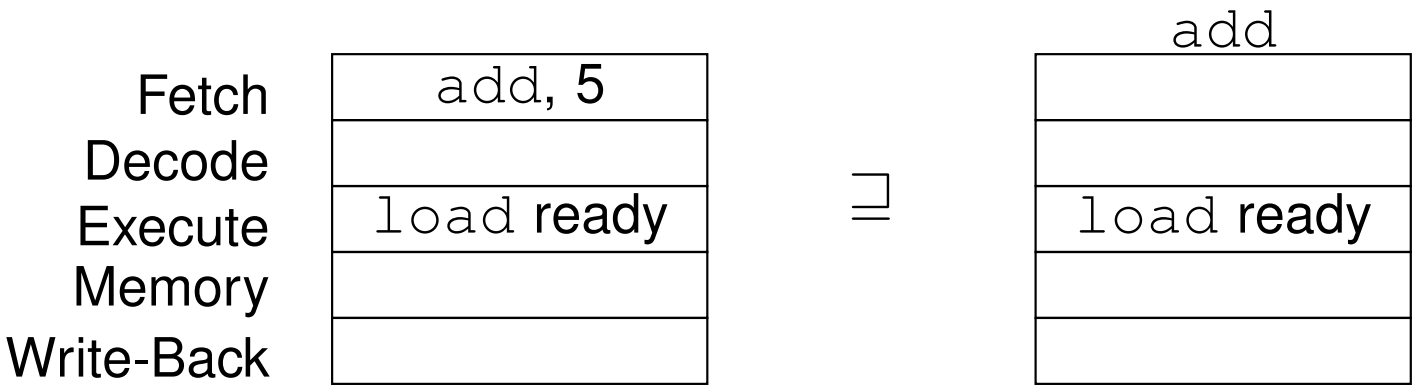
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline



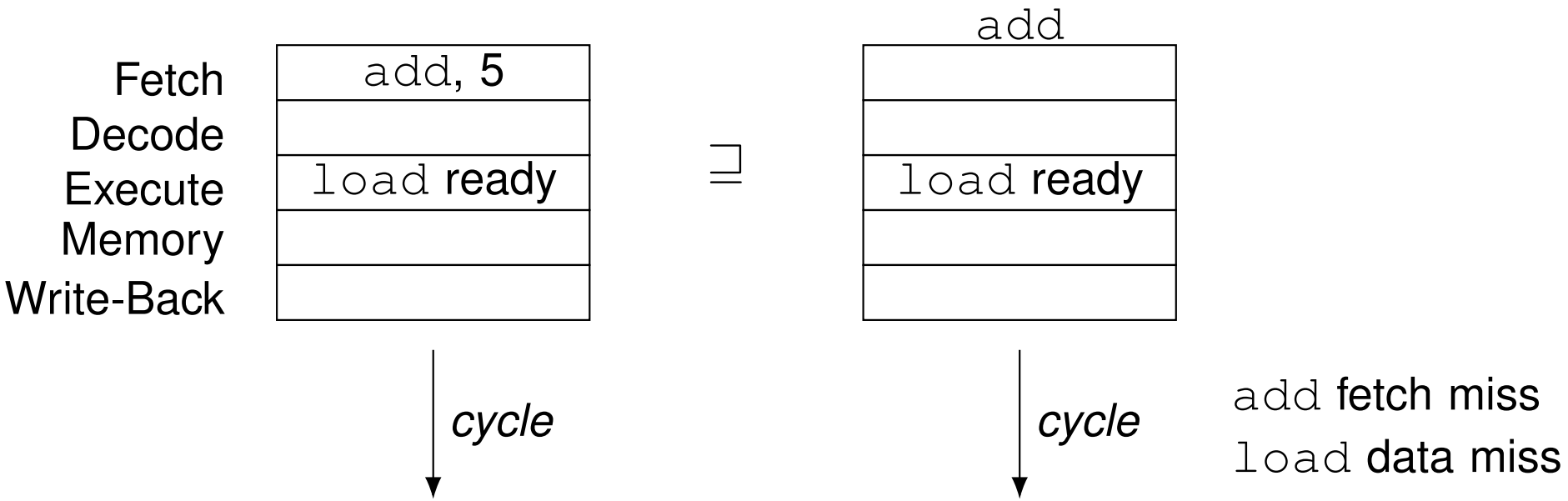
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline



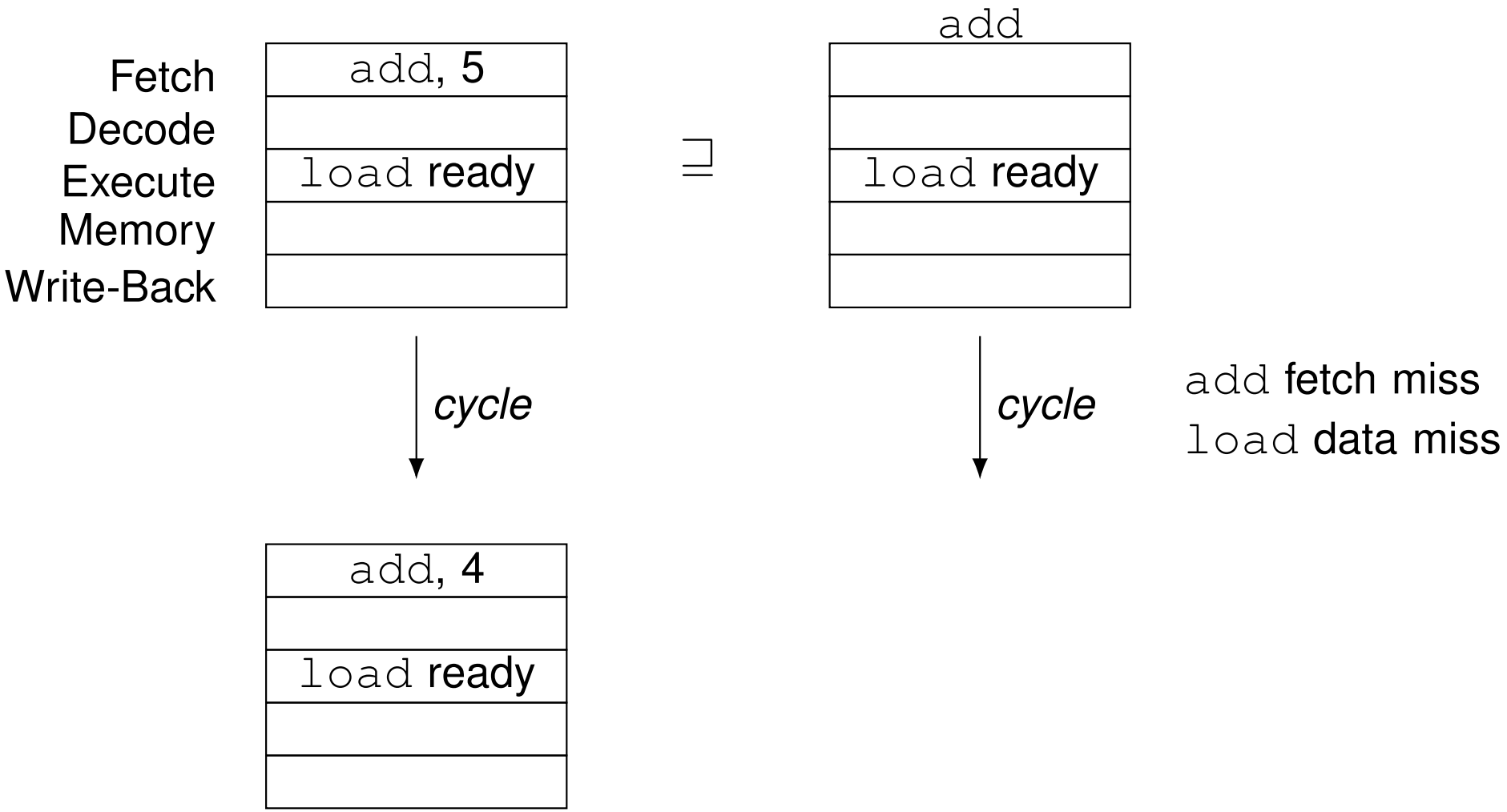
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline



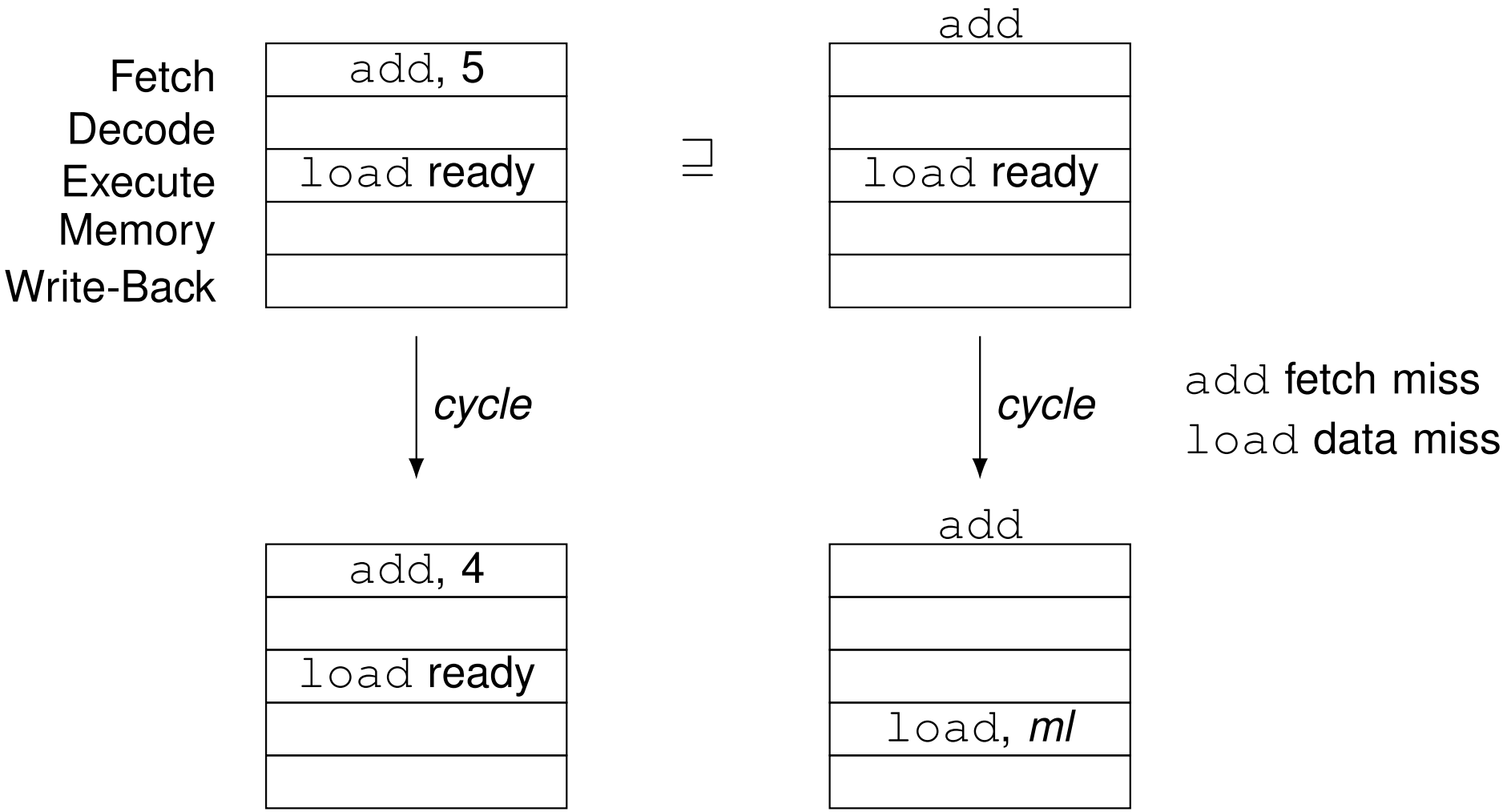
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline



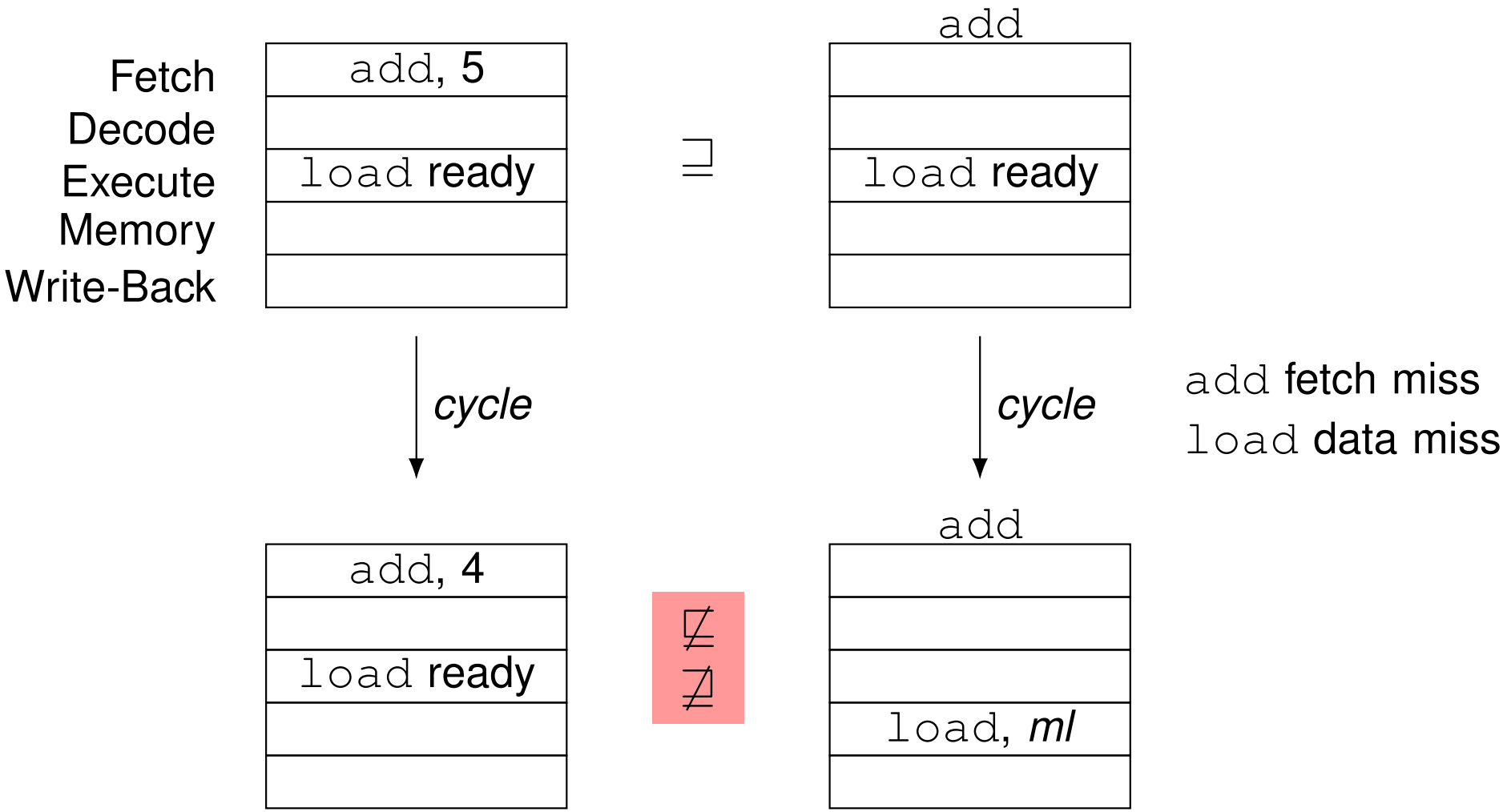
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline



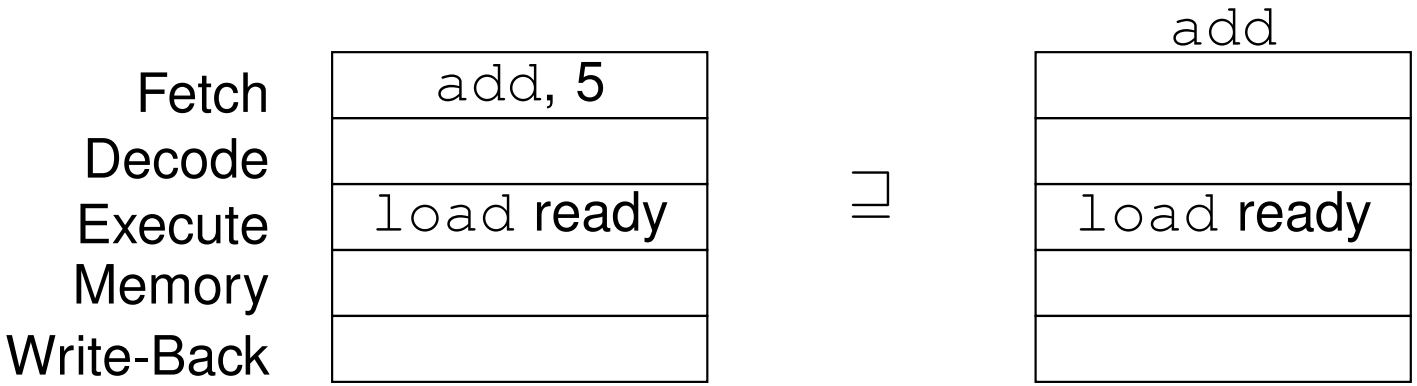
Even Simple Microarchitectures Behave Non-Monotonically

Conventional In-Order Pipeline



Even Simple Microarchitectures Behave Non-Monotonically

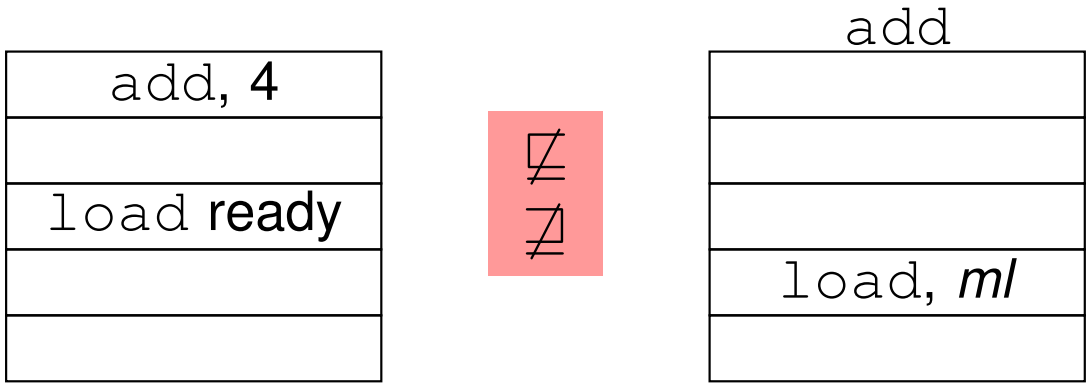
Conventional In-Order Pipeline



Relative progress of instructions influences order of bus accesses

miss

a miss



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

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

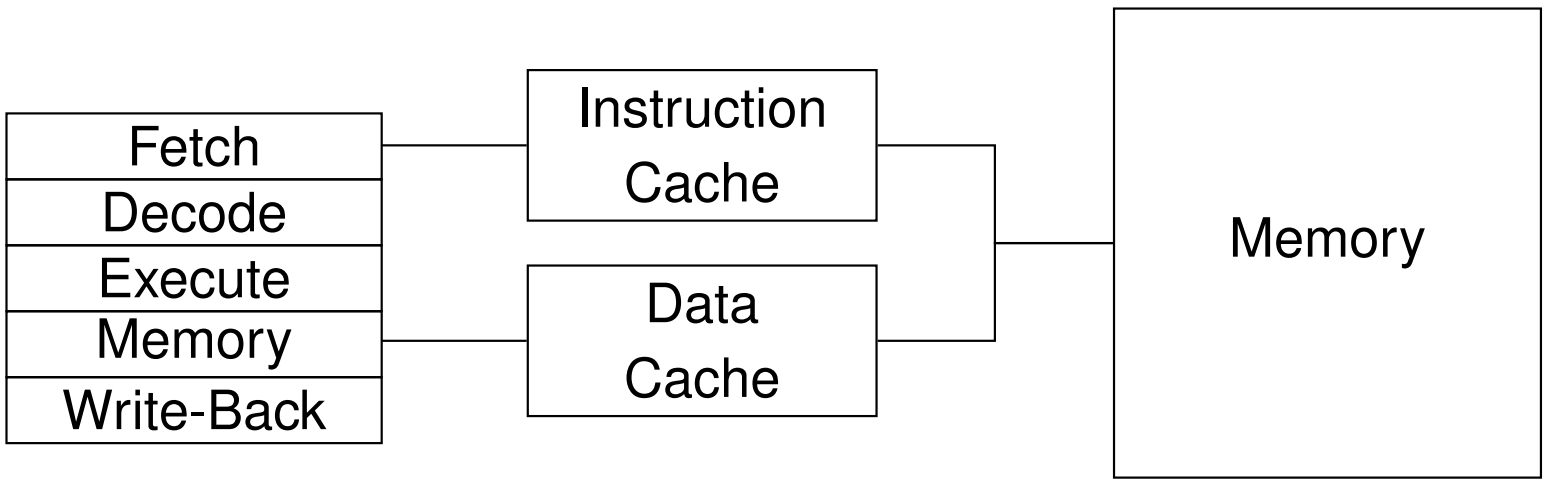
Enforce Monotonic Behaviour

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

SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

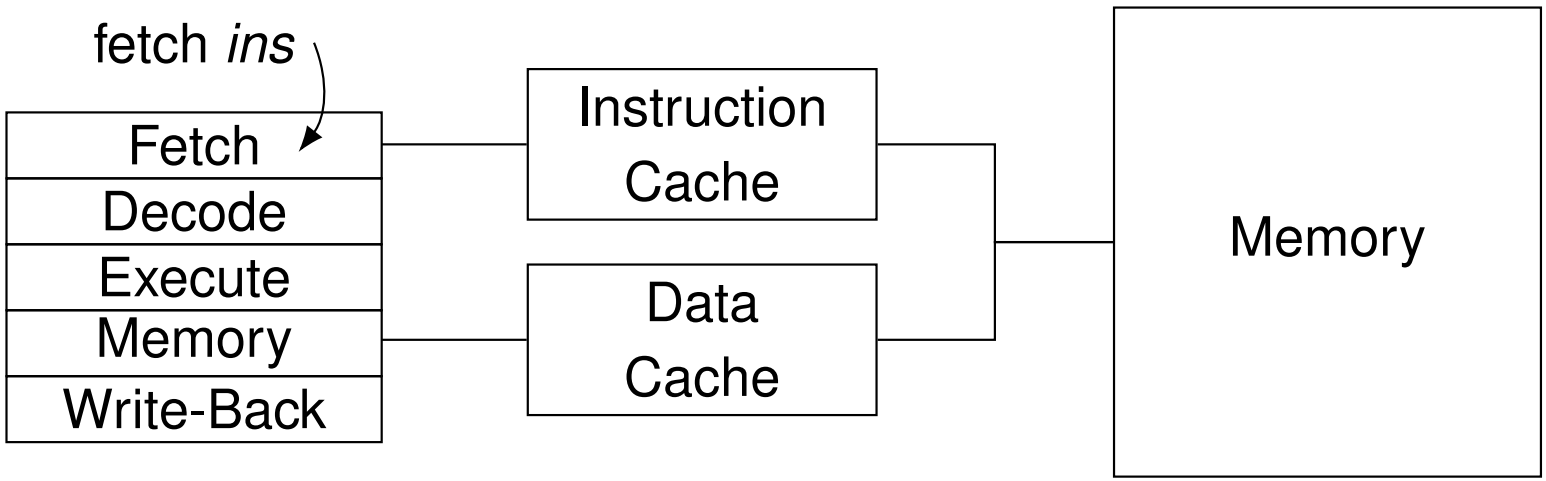
Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline



SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

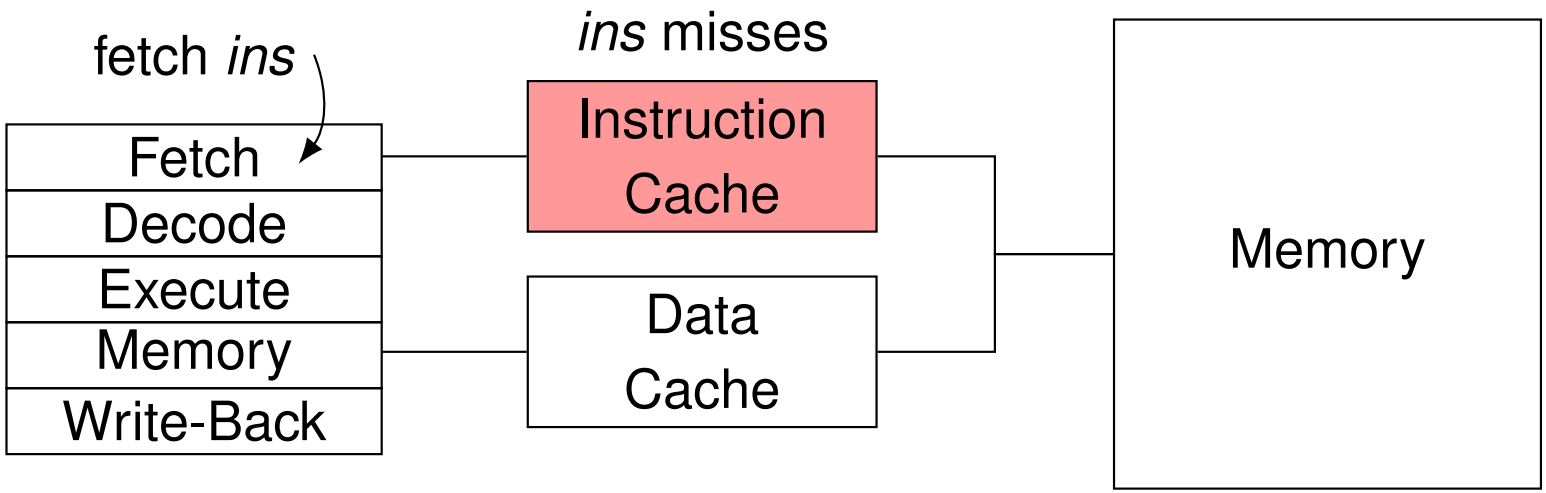
Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline



SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

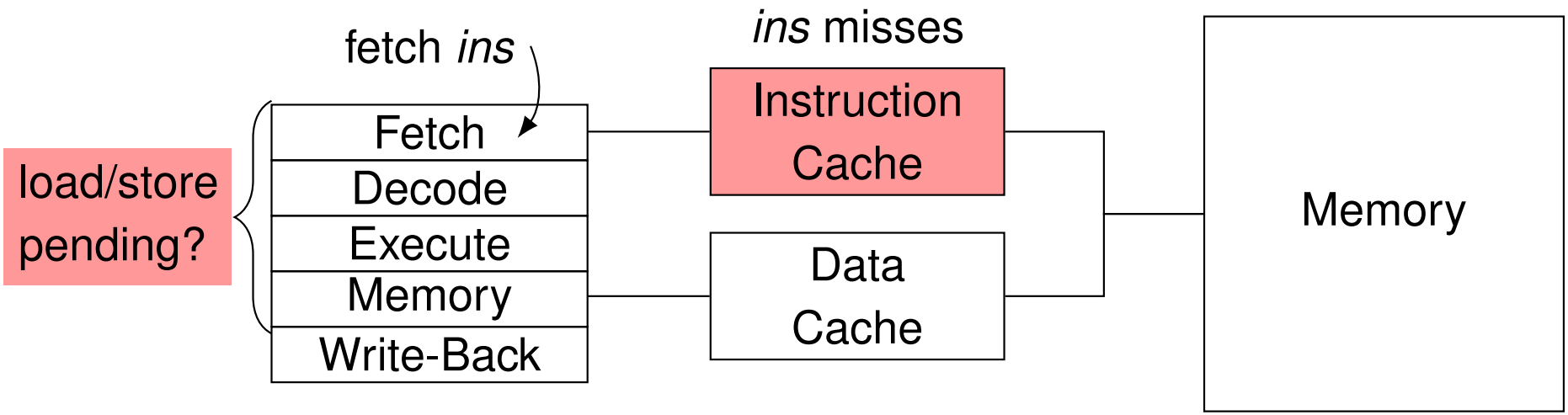
Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline



SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

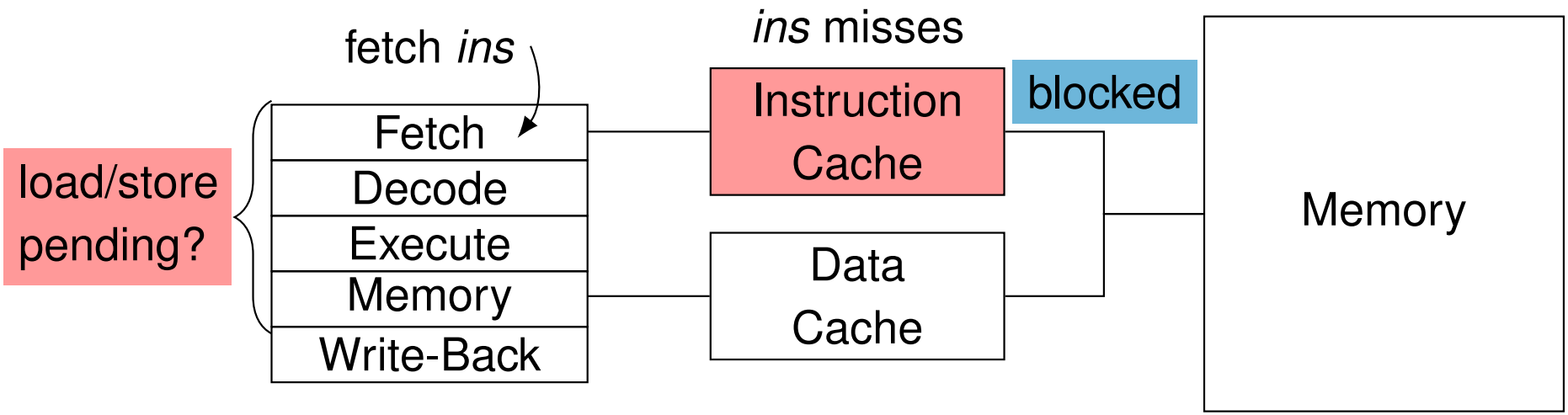
Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline



SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

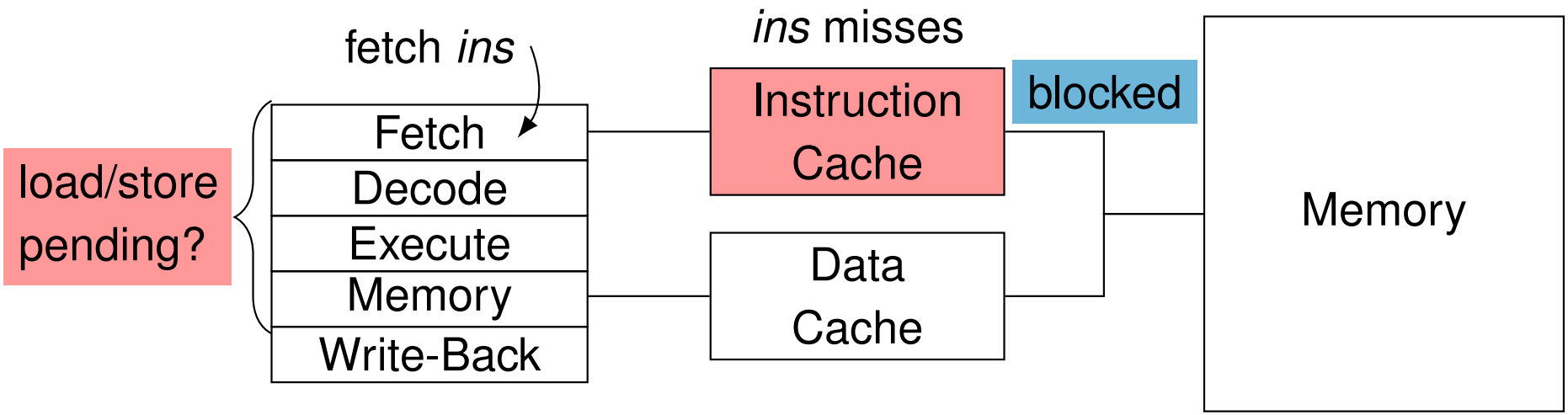
Implementation: Prevent bus accesses of instruction fetches while data-accessing instructions in the pipeline



SIC: Strictly In-Order Pipelined Core

Enforce Monotonic Behaviour

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



$\Rightarrow$ strict bus access order

In Our Paper

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, reineke}@cs.uni-saarland.de

***Abstract*—We introduce the strictly in-order core (SIC), a timing-predictable pipelined processor core. SIC is provably timing compositional and free of timing anomalies. This enables 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 pipelining 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 states 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 devise 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 highly pessimistic execution time bounds. The greatest analysis precision would be achieved by *fully-integrated timing analyses* [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 each 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.

1

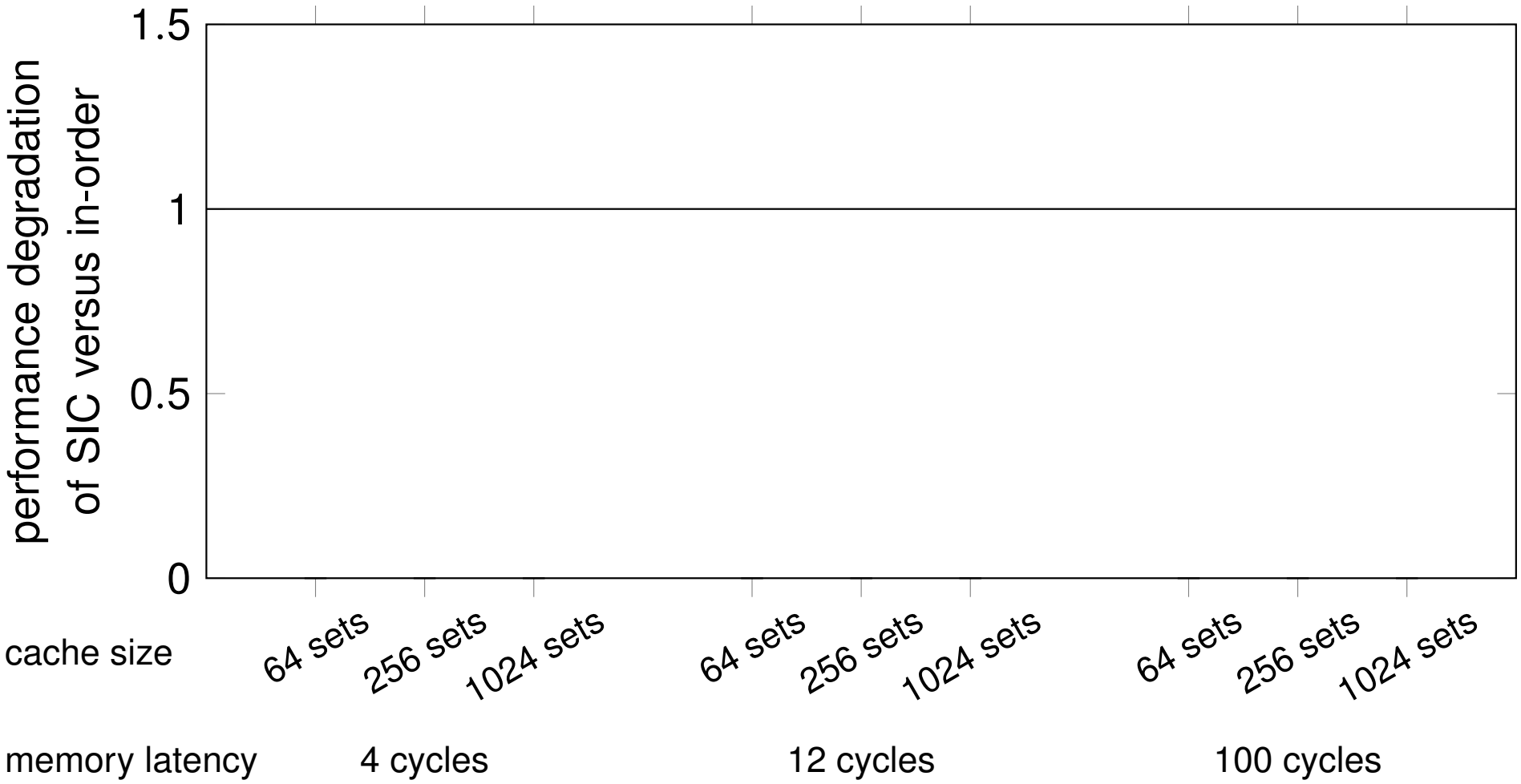
- Formal definition of progress
- Definition of strictly in-order behaviour
- Proof of monotonicity
- Corollary: anomaly freedom
- Corollary: timing compositionality

Evaluation Questions

1.) Performance Overhead of Enforcing Access Order

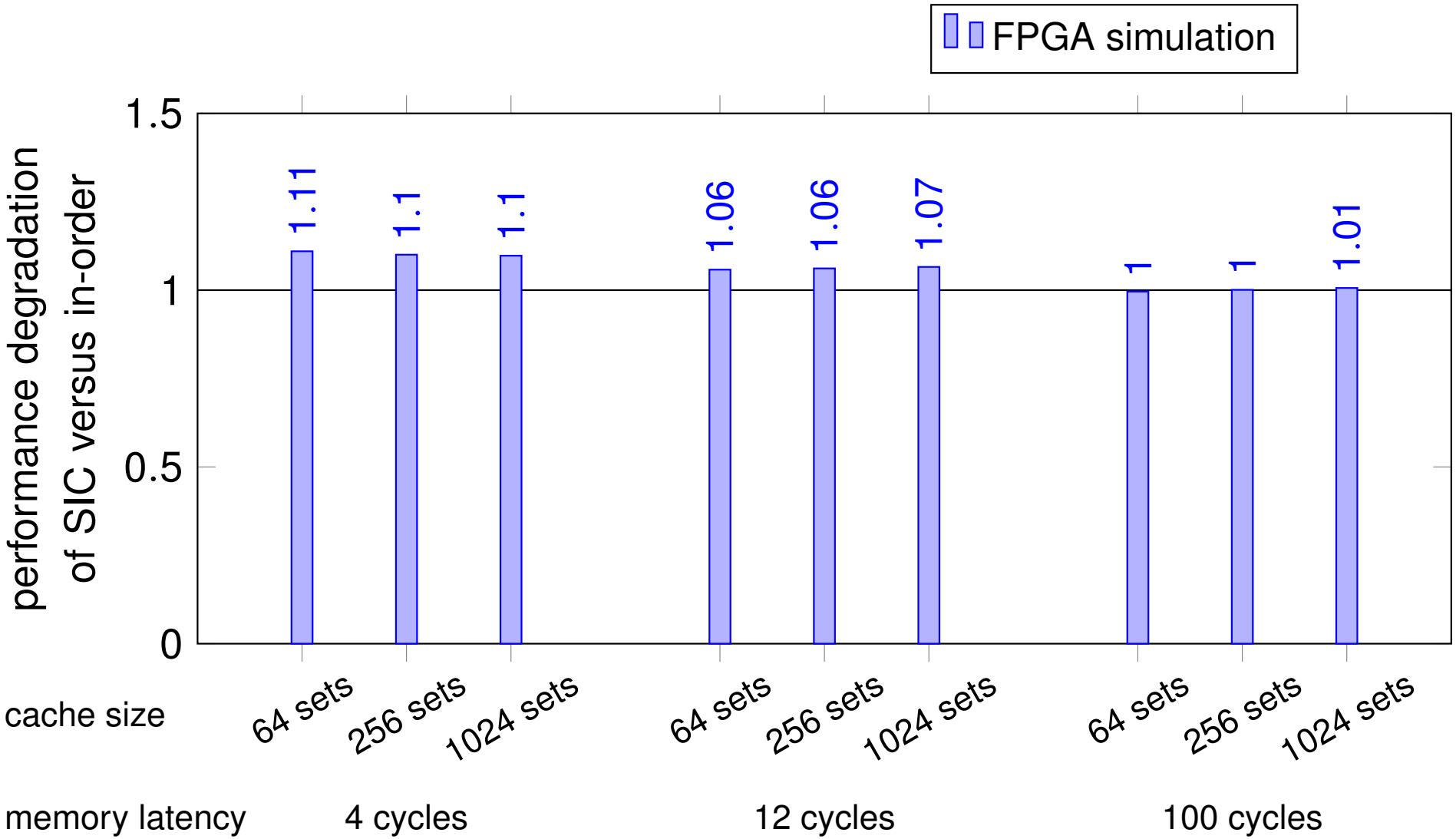
2.) Gain in Analysis Efficiency

Performance SIC versus Conventional In-Order

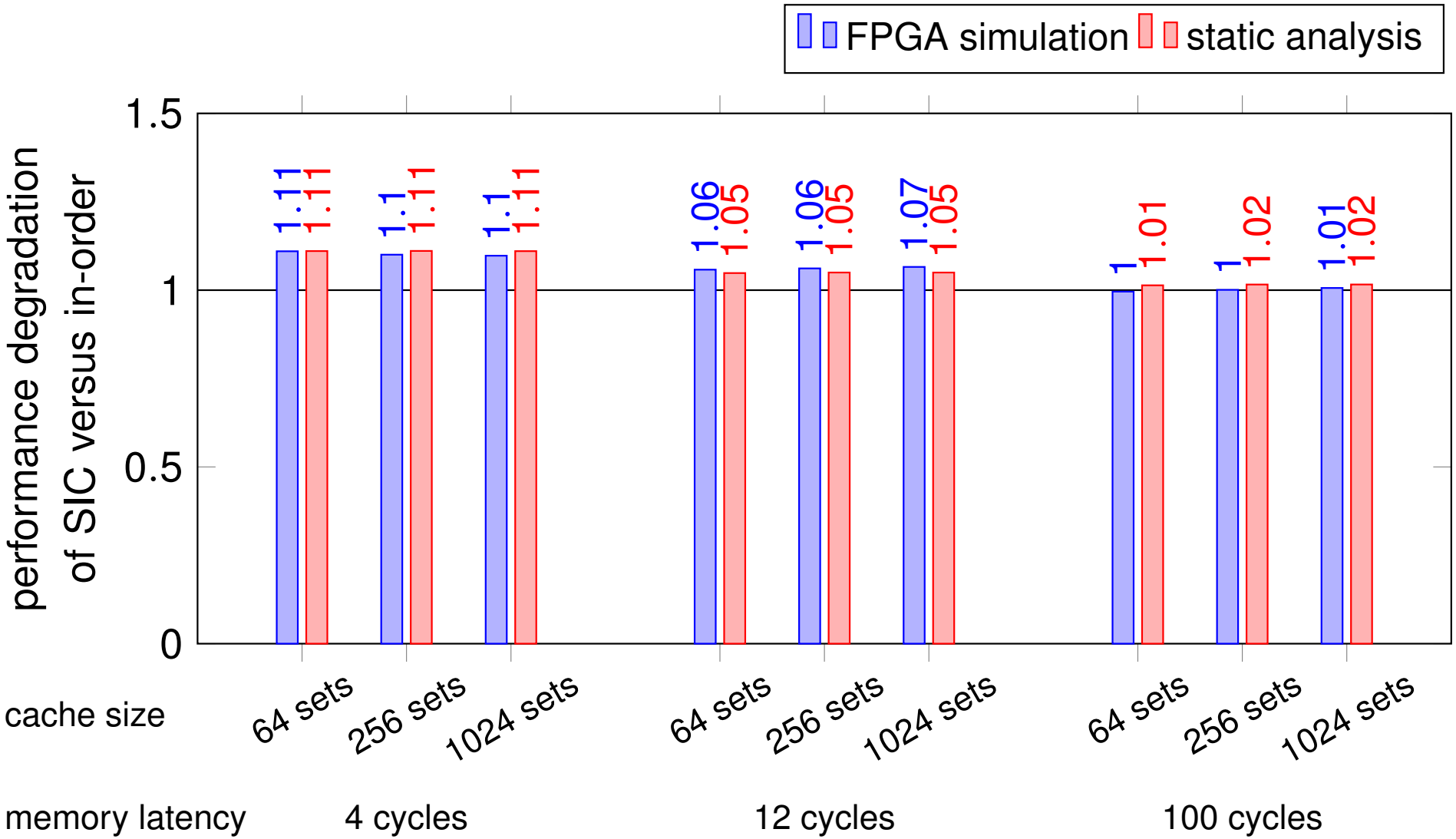


Performance SIC versus Conventional In-Order

FPGA implementation overhead 0.2%

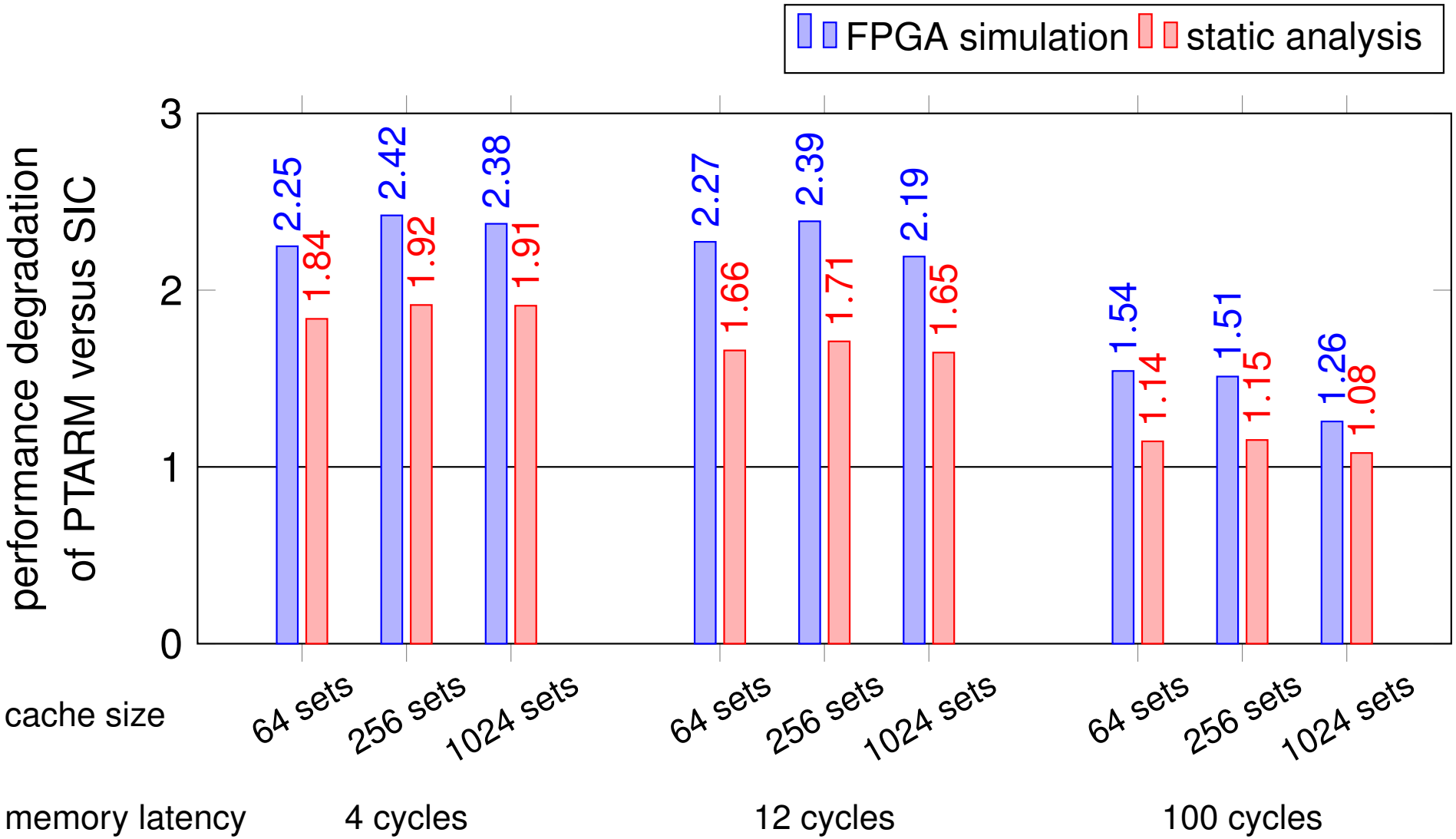


Performance SIC versus Conventional In-Order

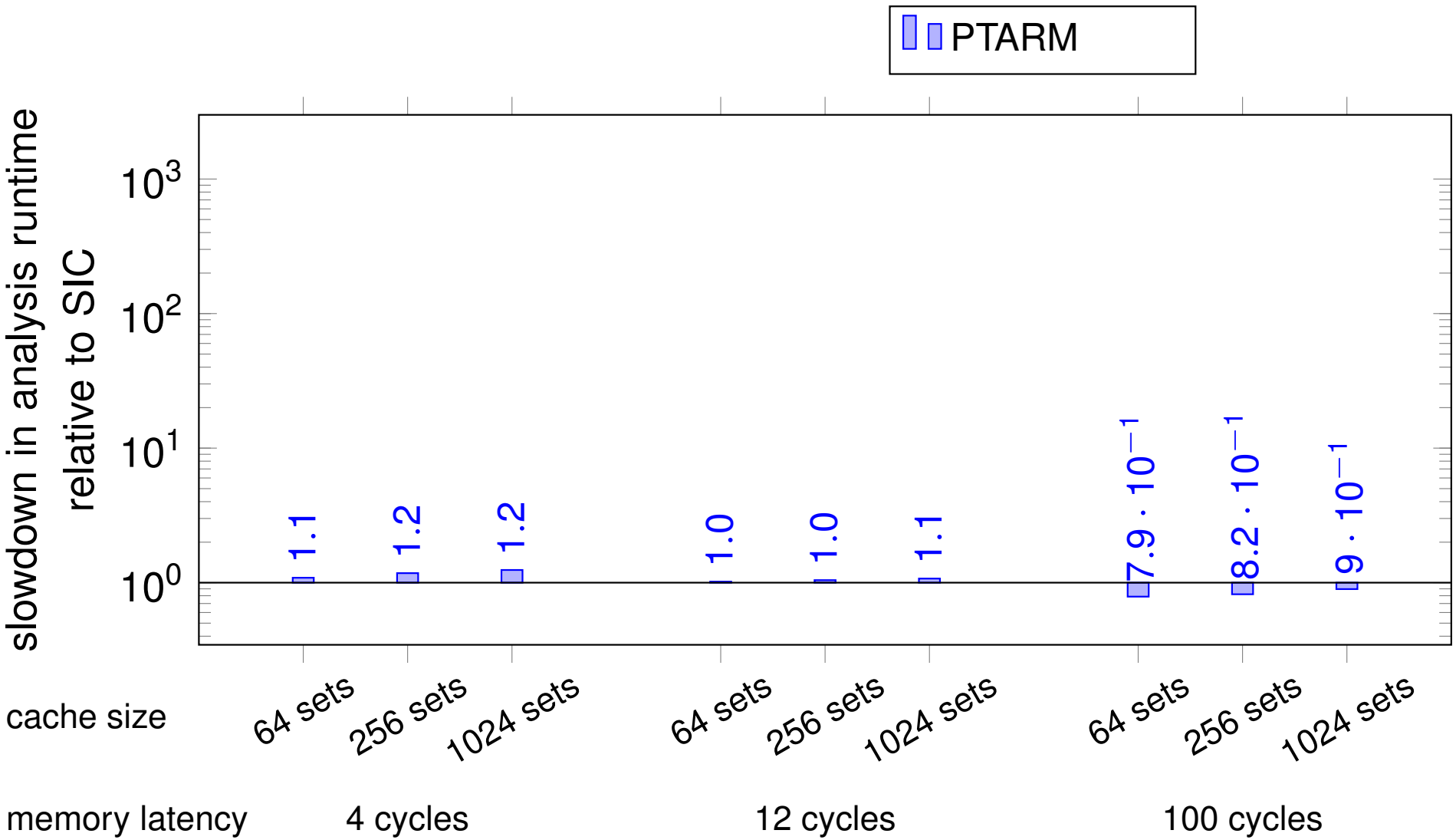


Performance Single PTARM-Thread versus SIC

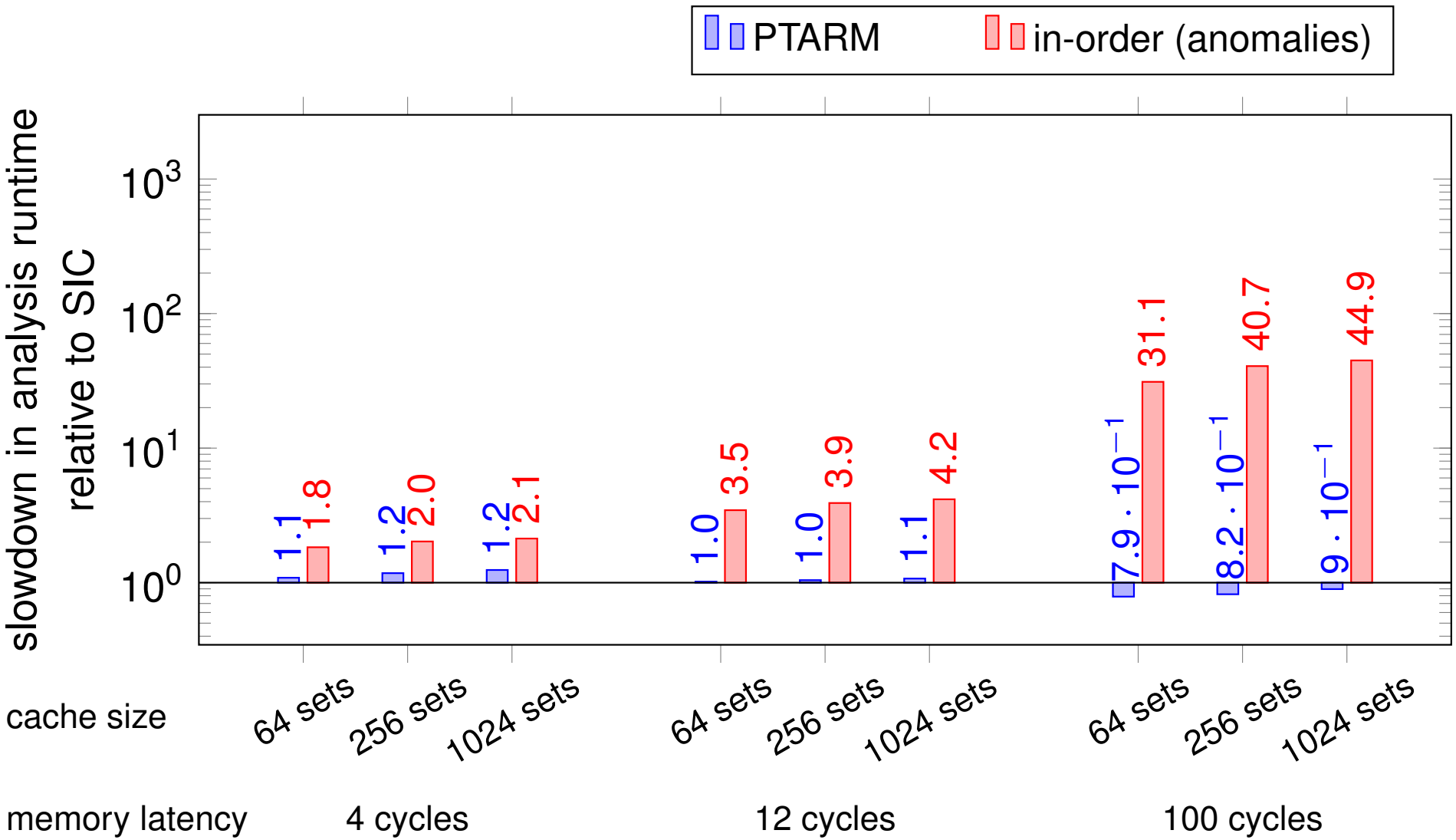
A Precision-Timed (PRET) Microarchitecture Implementation [Liu et al., ICCD'12]



Task-Level Analysis Cost

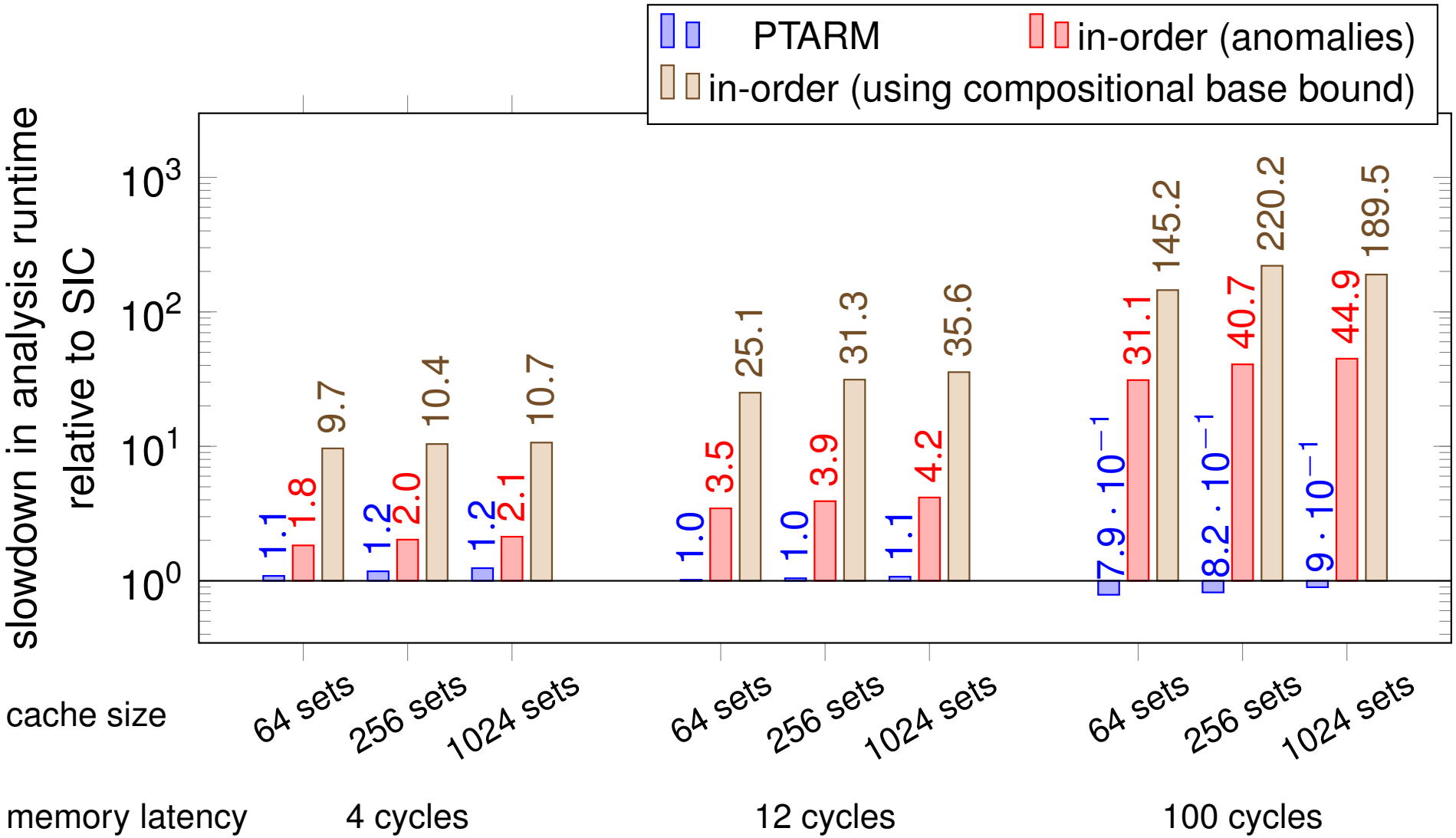


Task-Level Analysis Cost



Task-Level Analysis Cost

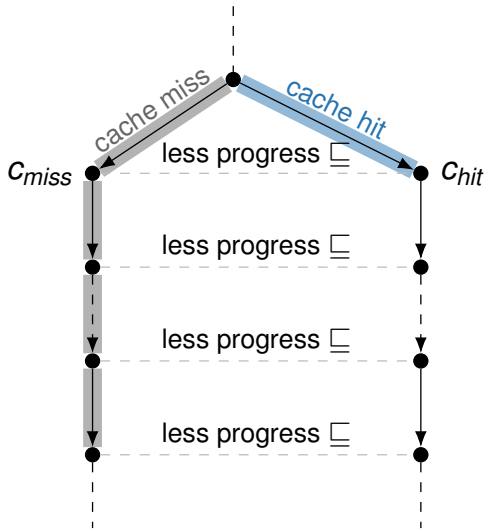
Compositional Base Bound [Hahn et al., RTNS'16]

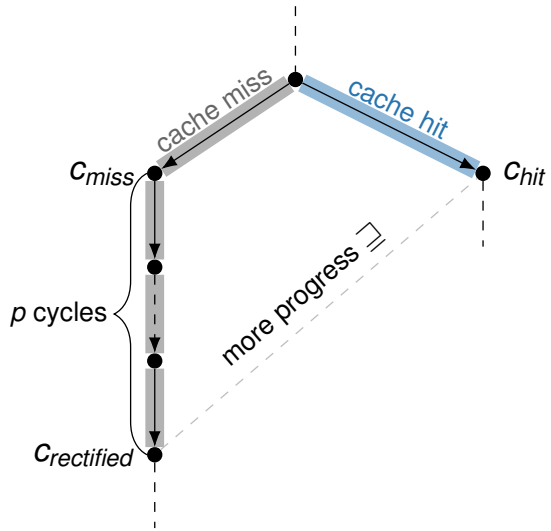


Conclusions

Strictly in-order pipeline

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





- yet performant
- practical: low implementation overhead