



# Timing Predictability and How to Achieve It

Jan Reineke @

SAARLAND  
UNIVERSITY



COMPUTER SCIENCE

Dagstuhl Seminar

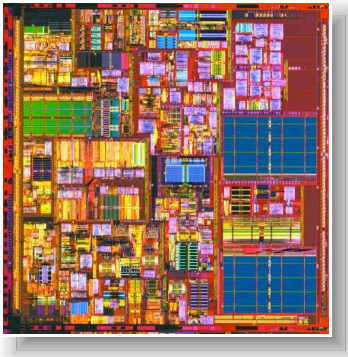
“Adaptive Isolation for Predictability and Security”

*October 31, 2016*

# The 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]);  
}
```

*Set of Software Tasks*



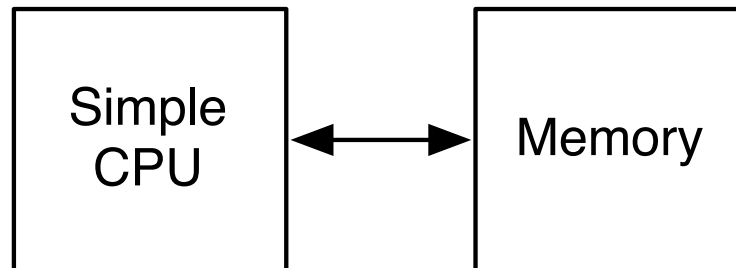
*Microarchitecture*



*Timing Requirements*

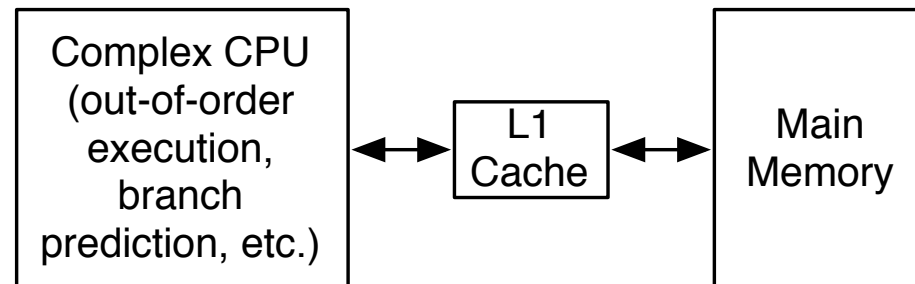
# What does the execution time depend on?

- 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.



# Example of Influence of Microarchitectural State

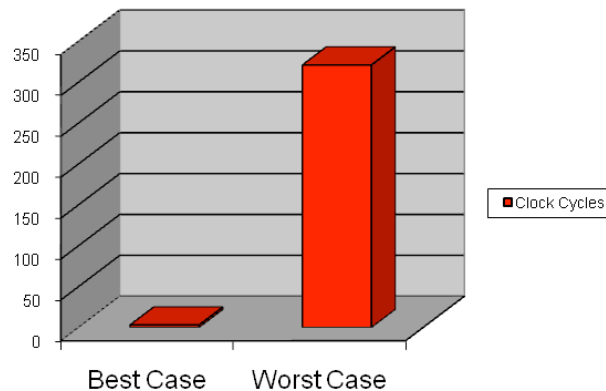
$x = a + b;$

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



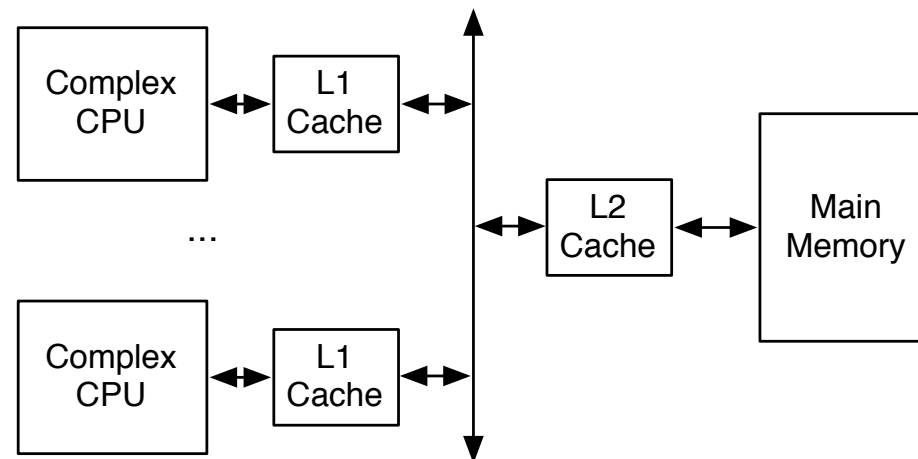
*PowerPC 755*

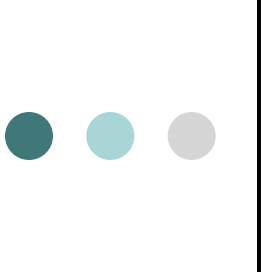
Execution Time (Clock Cycles)



# 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.





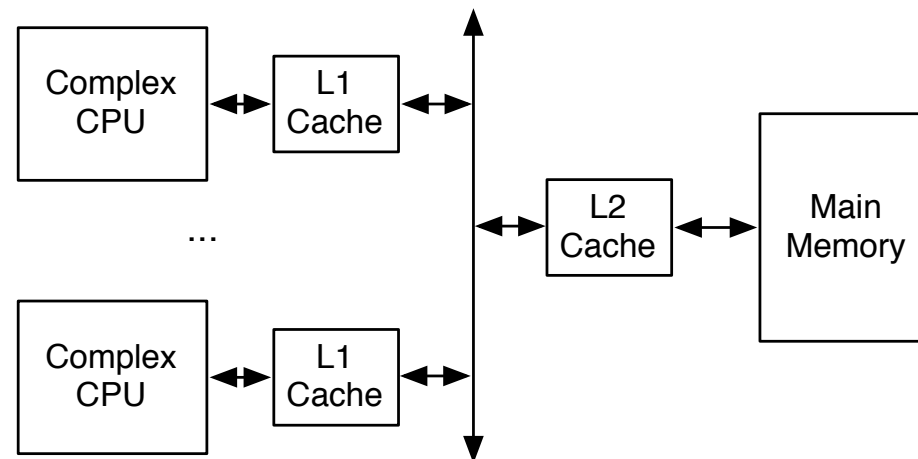
## Example of Influence of Corunning Tasks in Multicores

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**

# What does the execution time depend on?

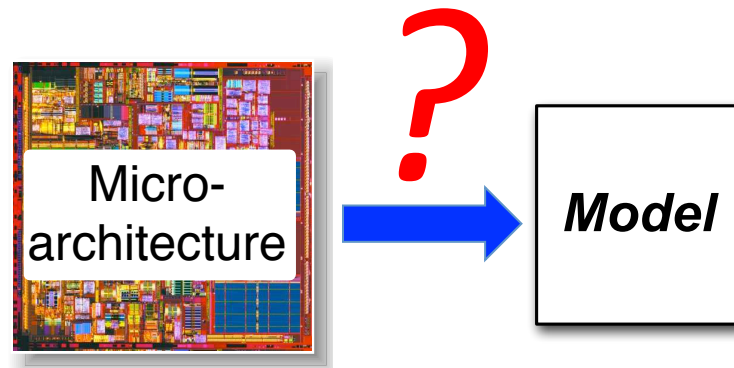
- The **input**, determining which path is taken through the program.
- The **state of the hardware platform**:
  - Due to caches, pipelining, speculation, etc.
- **Interference** from the environment:
  - External interference as seen from the analyzed task on shared busses, caches, memory.





# The Need for Models

Predictions about the future behavior of a system are always based on models of the system.



*All models are wrong, but some are useful.*

*George Box (Statistiker)*



# The Need for Timing Models

The ISA **partially** defines the behavior of microarchitectures: it **abstracts from timing**.

How to obtain **timing models**?

- Hardware manuals
- Manually devised microbenchmarks
- Machine learning

***Challenge:*** Introduce HW/SW contract to capture timing behavior of microarchitectures.



# Desirable Properties of Systems and their Timing Models

- Predictability
- Analyzability



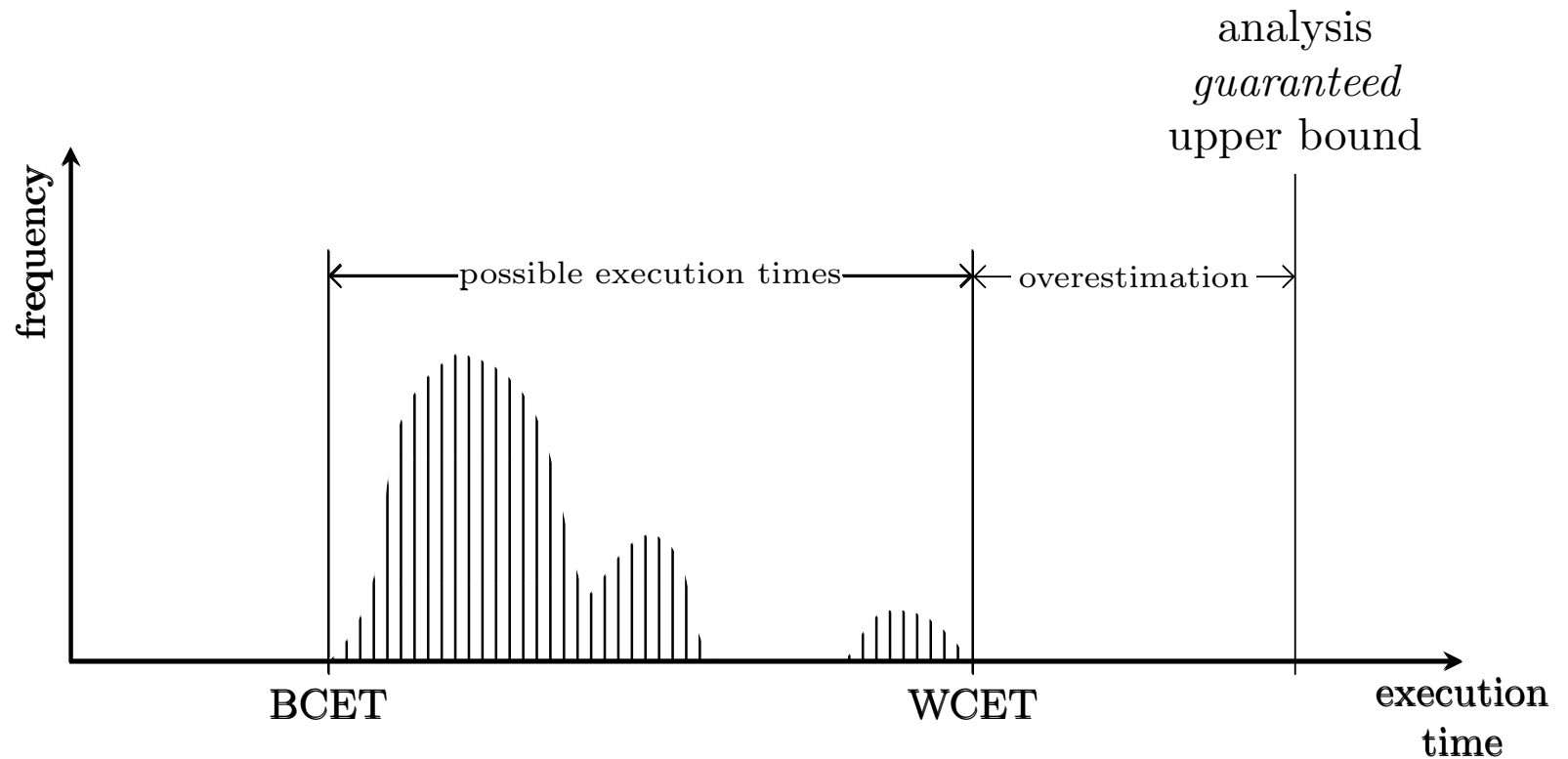
## Predictability

How **precisely** can programs' execution times on a particular microarchitecture be predicted?

Assuming a **deterministic** timing model and known initial conditions, can perfectly predict execution time.

But: initial state, inputs, and interference unknown.

# Timing Predictability





# How to Increase Predictability?

## 1. *Eliminate stateful components:*

~~Cache~~ → Scratchpad memory

~~Regular Pipeline~~ → Thread-interleaved pipeline

~~Out-of-order execution~~ → VLIW

***Challenge: Efficient static allocation of resources.***



## How to Increase Predictability?

### *2. Eliminate interference: „temporal isolation“*

Partition resources:

*in time* {

- TDMA bus/NoC arbitration,
- SW scheduling (e.g. PREM)

*in space* {

- shared cache: in HW or SW
- SRAM banks (e.g. Kalray MPPA)
- DRAM banks (e.g. PRET DRAM, PALLOC)

***Challenge:***

*Determine efficient partitioning of resources.*

***Question: What's the performance impact?***



## How to Increase Predictability?

3. Choose „forgetful“/„insensitive“ components:  
FIFO, PLRU replacement → LRU replacement  
[Real-Time Systems 2007, WAOA 2015]

### ***Open Problems:***

- *Is there a systematic way to design „forgetful“ microarchitectural components?*
- *Can randomization help?*





## Analyzability

How ***efficiently*** can programs' WCETs on a particular microarchitecture be bounded?

WCET analysis needs to consider all inputs, initial HW states, interference scenarios...

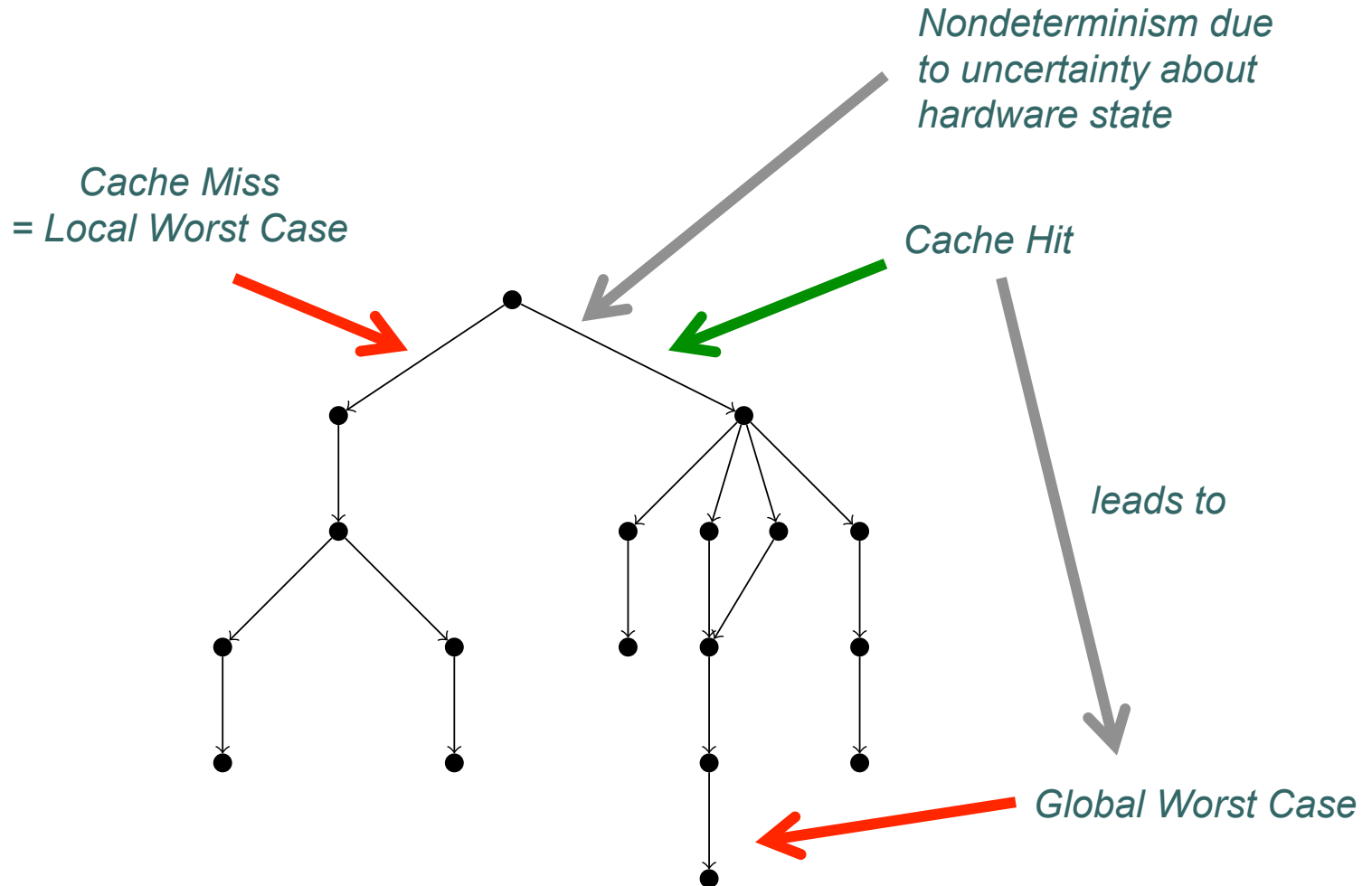
...explicitly or implicitly.



# How to Increase Analyzability?

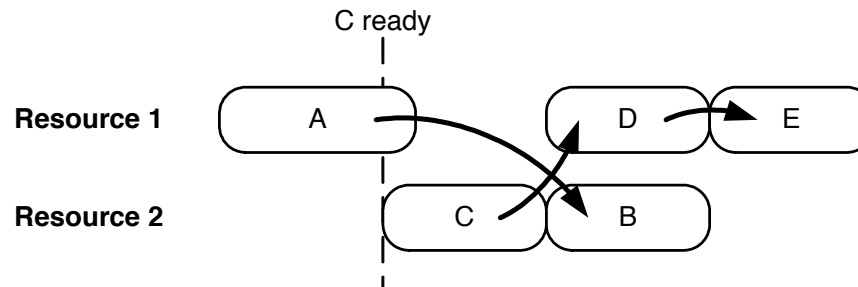
1. Eliminate stateful resources:  
Fewer states to consider
2. Eliminate interference: „temporal isolation“:  
Can focus analysis on one partition
3. Choose „forgetful“/“insensitive“ components:  
Different analysis states will quickly converge
4. Enable efficient **implicit** treatment of states:
  - Monotonicity / Freedom from Timing Anomalies
  - Timing Compositionality

# Timing Anomalies



# Timing Anomalies: Example

## Scheduling Anomaly

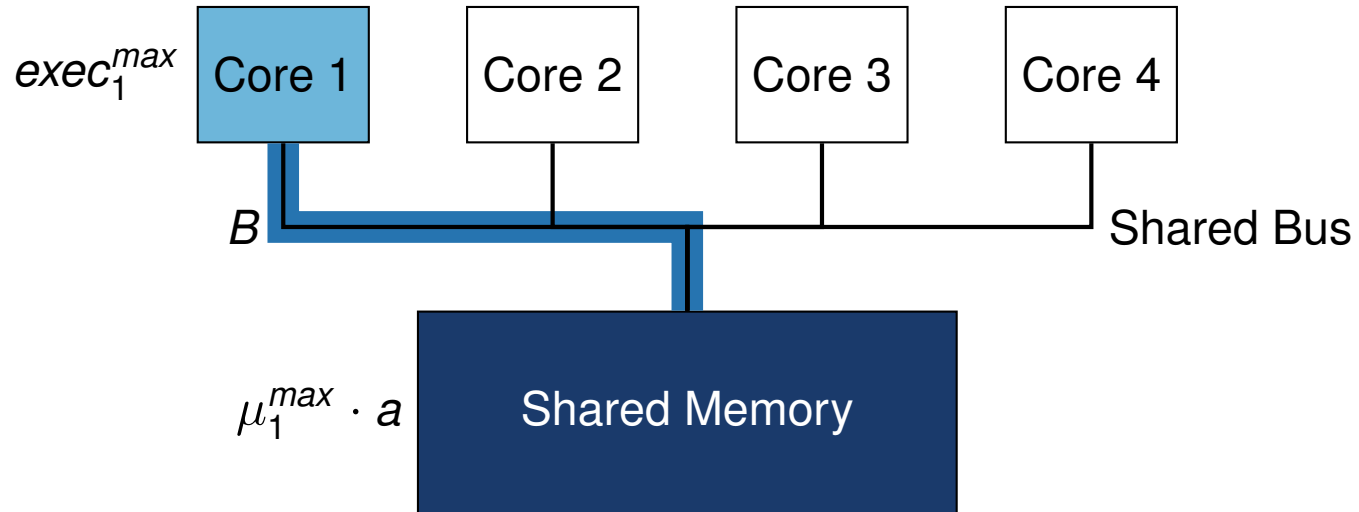


*Bounds on multiprocessing timing anomalies*

*RL Graham - SIAM Journal on Applied Mathematics, 1969 – SIAM*

*(<http://epubs.siam.org/doi/abs/10.1137/0117039>)*

# Timing Compositionality: By Example

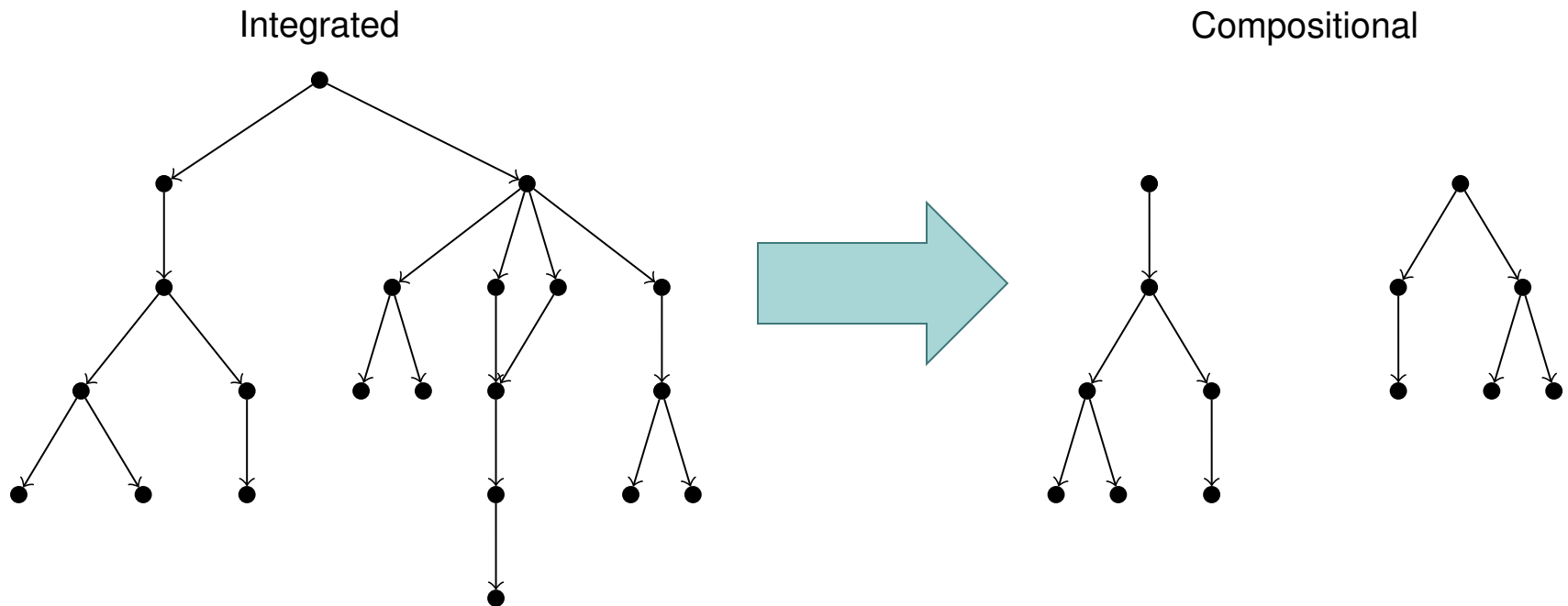


**Timing Compositionality** =

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



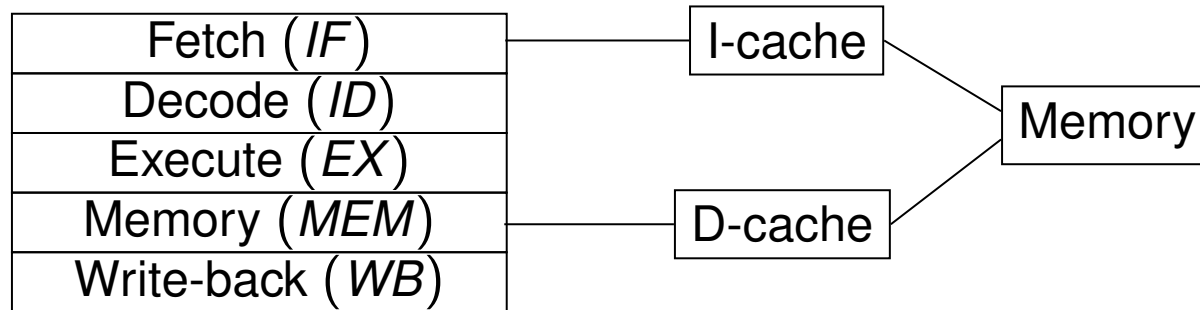


# Conventional Wisdom

Simple in-order pipeline + LRU caches

- no timing anomalies
- timing-compositional

# Bad News I: Timing Anomalies



We show such a pipeline has timing anomalies:

***Toward Compact Abstractions for Processor Pipelines***

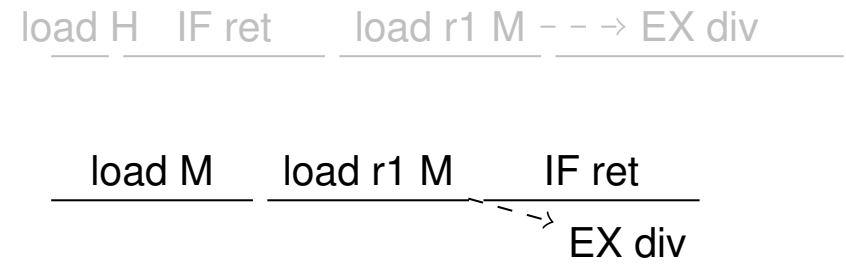
*S. Hahn, J. Reineke, and R. Wilhelm. In Correct System Design, 2015.*



# A Timing Anomaly

```
load ...
nop
load r1, ...
div ..., r1
-----
ret
```

(load r1, 0)
(load, 0)



## Hit case:

- Instruction fetch starts before second load becomes ready

## Intuitive Reason:

Progress in the pipeline influences order of instruction fetch and data access

- Second load is prioritized over instruction fetch
- Loading before fetching suits subsequent execution



## Bad News II: Timing Compositionality

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



## Good News

Two approaches to solve problem:

1. Stall entire processor upon „timing accidents“
2. Strictly in-order pipeline



# Strictly In-Order Pipelines: Definition

## *Definition (Strictly In-Order):*

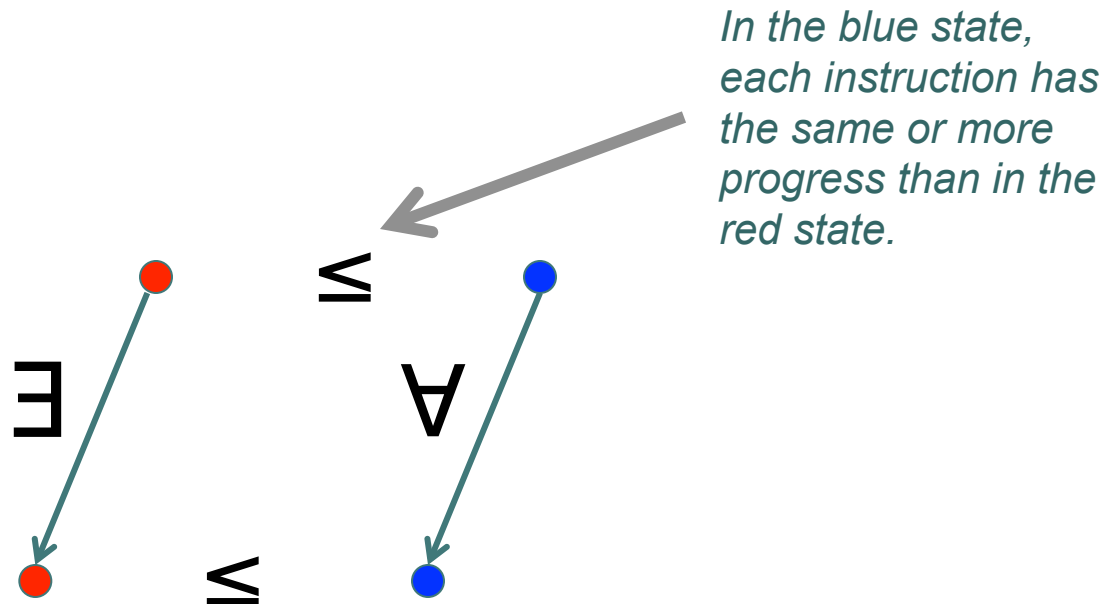
We call a pipeline *strictly in-order* if each *resource* processes the instructions in program order.

- Enforce memory operations (instructions and data) in-order (common memory as resource)
- Block instruction fetch until no potential data accesses in the pipeline

# Strictly In-Order Pipelines: Properties

## *Theorem 1 (Monotonicity):*

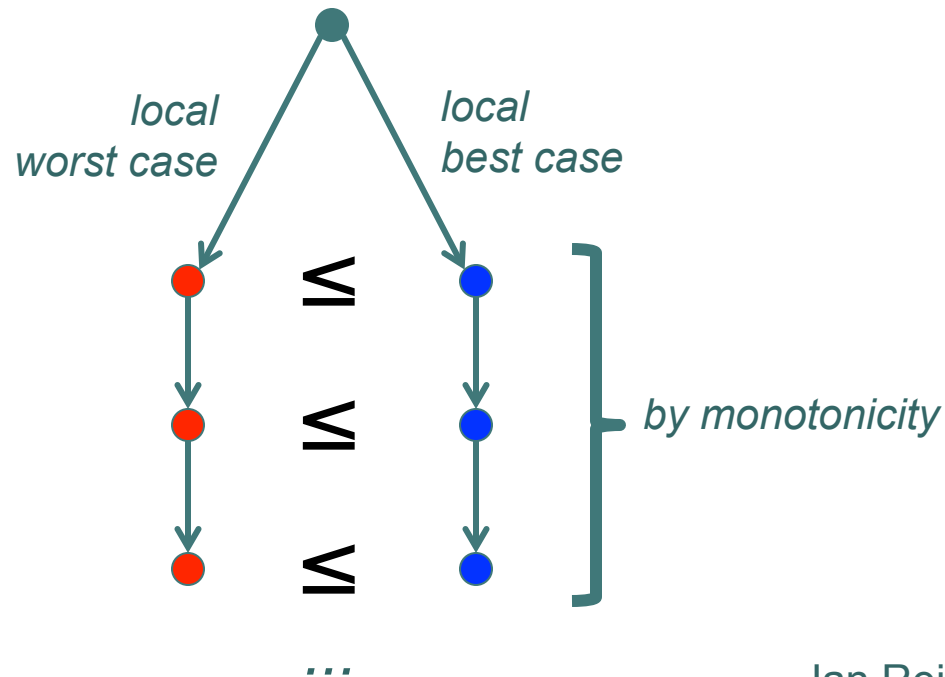
In the strictly in-order pipeline progress of an instruction is monotone in the progress of other instructions.



# Strictly In-Order Pipelines: Properties

## *Theorem 2 (Timing Anomalies):*

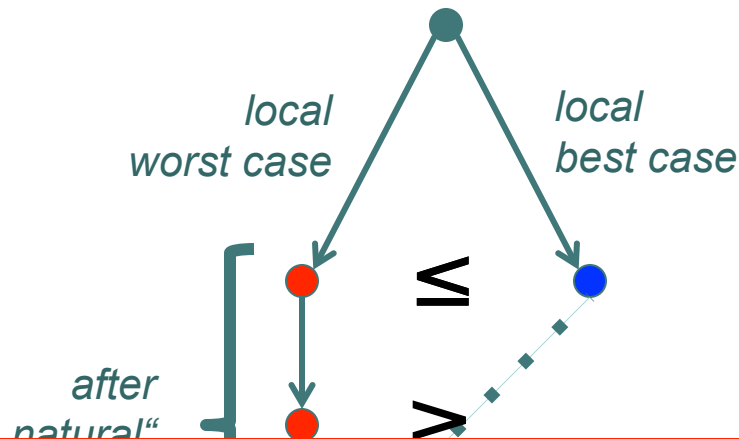
The strictly in-order pipeline is free of timing anomalies.



# Strictly In-Order Pipelines: Properties

*Theorem 3 (Timing Compositionality):*

The strictly in-order pipeline admits „compositional analysis with intuitive penalties.“



**Open Question:** What's the performance impact of being strictly in-order?



## Conclusions

- Need faithful timing models
- Various approaches to achieve predictability
  - Achieving efficiency is hard
- How to convince industry to build and to buy predictable processors?

*Thank you for your attention!*





# Some References

***Enabling Compositionality for Multicore Timing Analysis***

S. Hahn, M. Jacobs, and J. Reineke. In RTNS, 2016.

***MIRROR: Symmetric Timing Analysis for Real-Time Tasks on Multicore Platforms with Shared Resources***

W.-H. Huang, J.-J. Chen, and J. Reineke. In DAC, 2016.

***A Generic and Compositional Framework for Multicore Response Time Analysis***

S. Altmeyer, R.I. Davis, L.S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke. In RTNS, 2015.

***On the Smoothness of Paging Algorithms***

J. Reineke and A. Salinger. In WAOA, 2015.

***Toward Compact Abstractions for Processor Pipelines***

S. Hahn, J. Reineke, and R. Wilhelm. In Correct System Design, 2015.

***Architecture-Parametric Timing Analysis***

J. Reineke and J. Doerfert. In RTAS, 2014.

***Selfish-LRU: Preemption-Aware Caching for Predictability and Performance***

J. Reineke, S. Altmeyer, D. Grund, S. Hahn, C. Maiza. In RTAS, 2014.

***Towards Compositionality in Execution Time Analysis - Definition and Challenges***

S. Hahn, J. Reineke, and R. Wilhelm. In CRTS, 2013.

***Measurement-based Modeling of the Cache Replacement Policy***

A. Abel and J. Reineke. In RTAS, 2013.

***PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation***

J. Reineke, I. Liu, H.D. Patel, S. Kim, and E.A. Lee. In CODES+ISSS, 2011.

***Timing Predictability of Cache Replacement Policies***

J. Reineke, D. Grund, C. Berg, and R. Wilhelm. In Real-Time Systems, 2007.