Design and Analysis of Time-Critical Systems

Timing Predictability and Analyzability + Case Studies: PTARM and Kalray MPPA-256

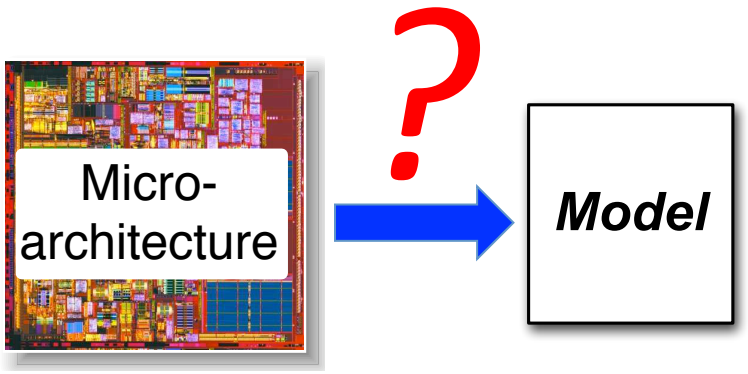
Jan Reineke @ SAARLAND UNIVERSITY
COMPUTER SCIENCE

ACACES Summer School 2017

Fiuggi, Italy

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

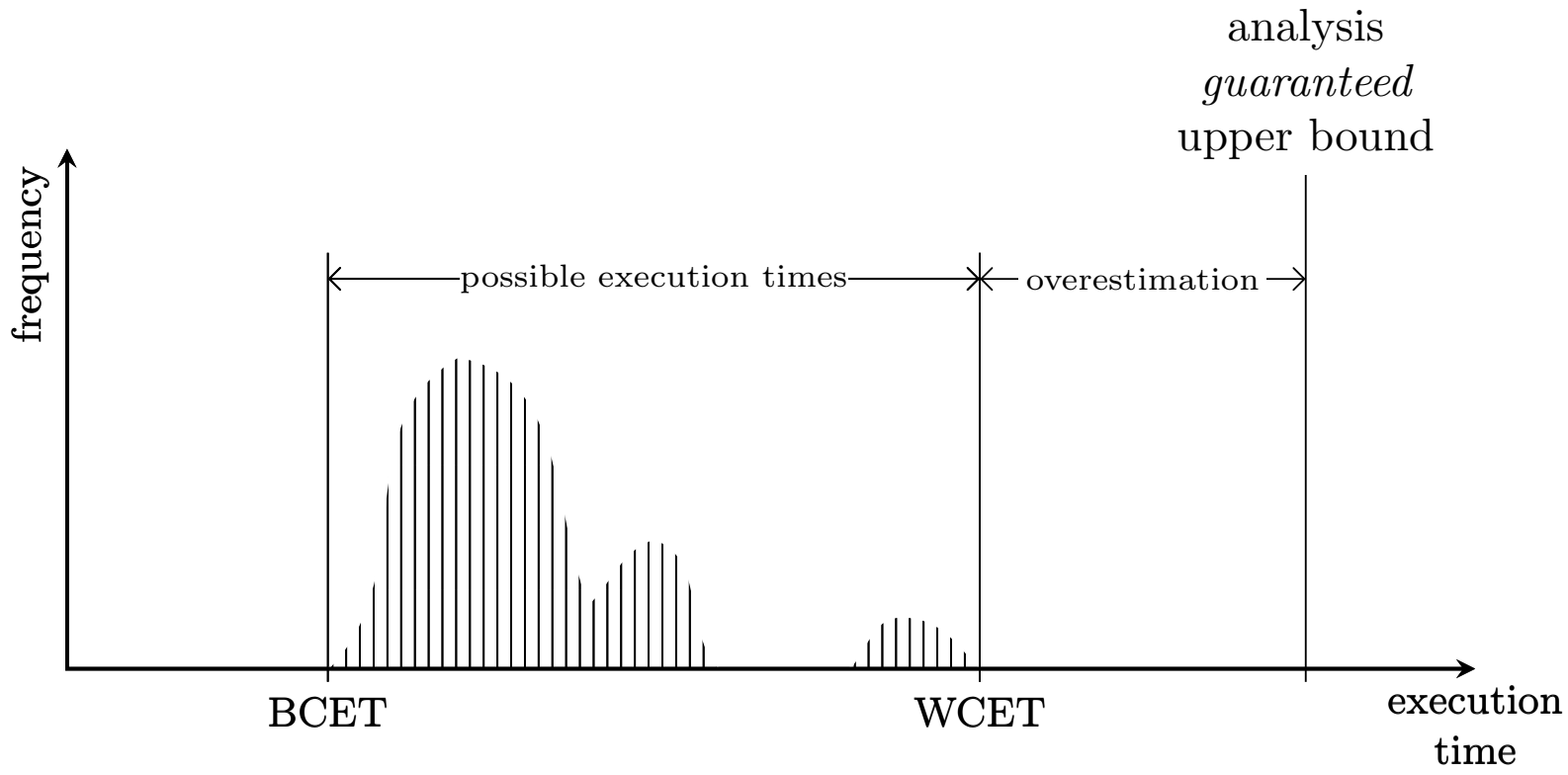
Predictability

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.

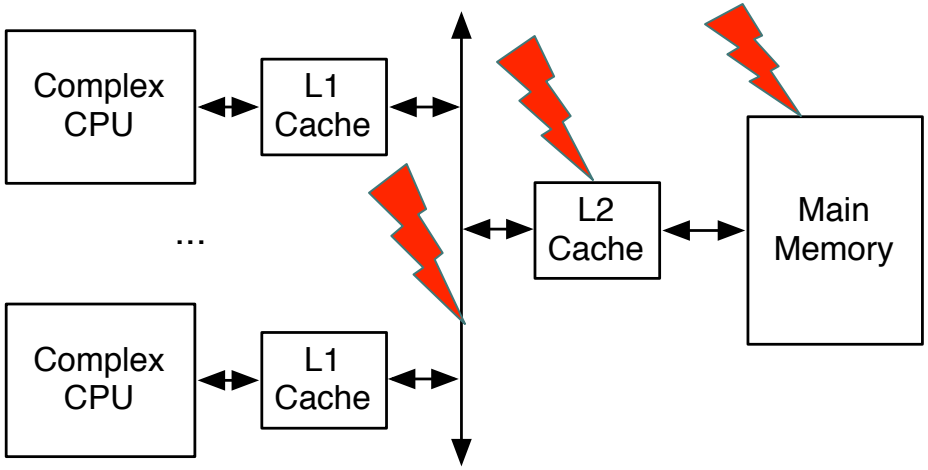
But: initial state, inputs, and interference unknown.

Timing Predictability



Reminder: 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.



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 Problem**:*
Is there a systematic way to design „forgetful" microarchitectural components?

Analyzability

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

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

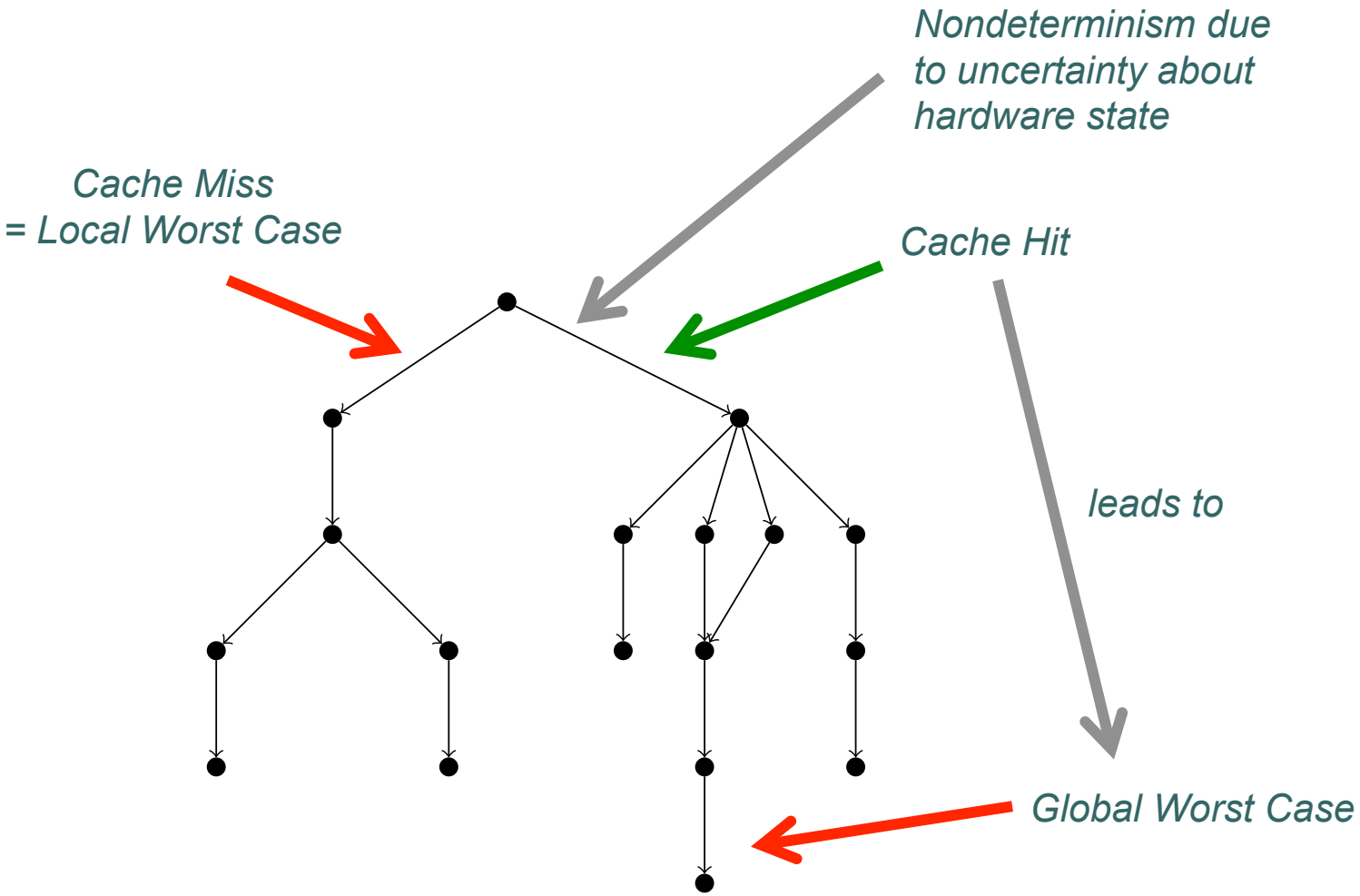
...explicitly or implicitly.

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

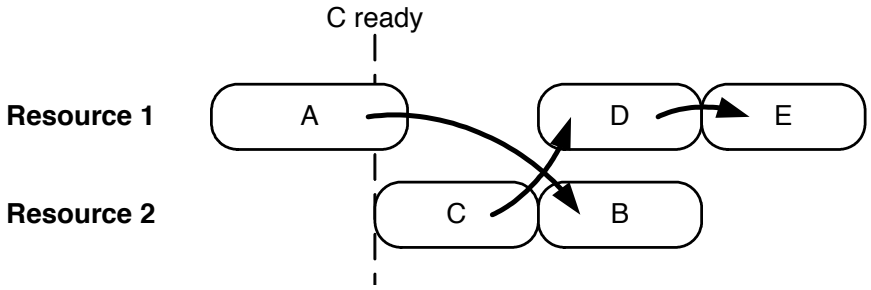
Timing Anomalies



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

Timing Anomalies: Example

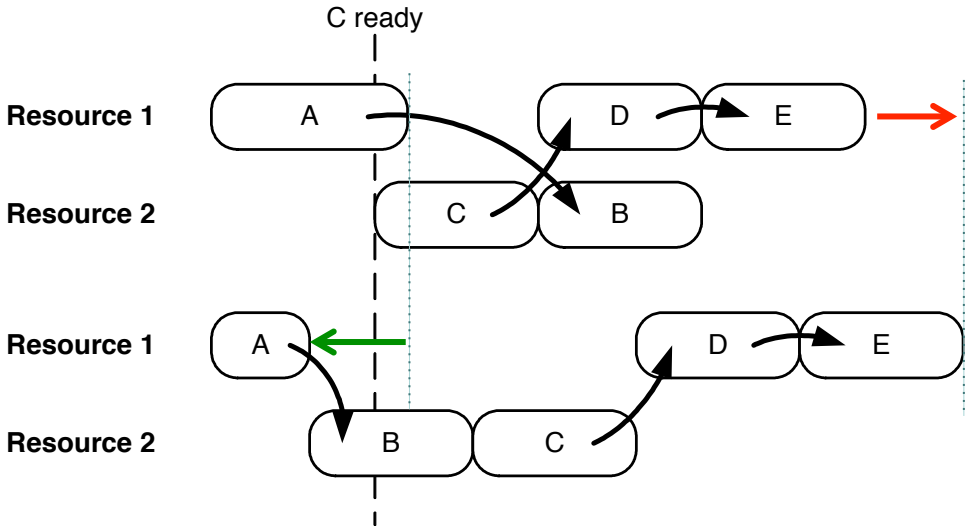
Scheduling Anomaly



Bounds on multiprocessing timing anomalies
RL Graham - SIAM Journal on Applied Mathematics, 1969 – SIAM
(http://epubs.siam.org/doi/abs/10.1137/0117039)

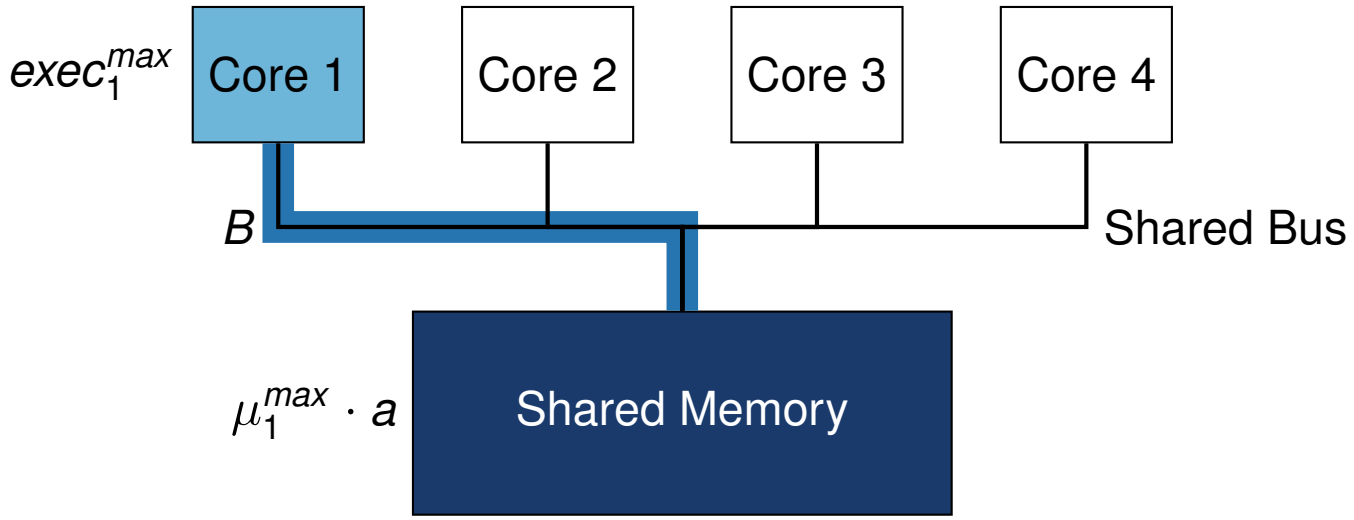
Timing Anomalies: Example

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)

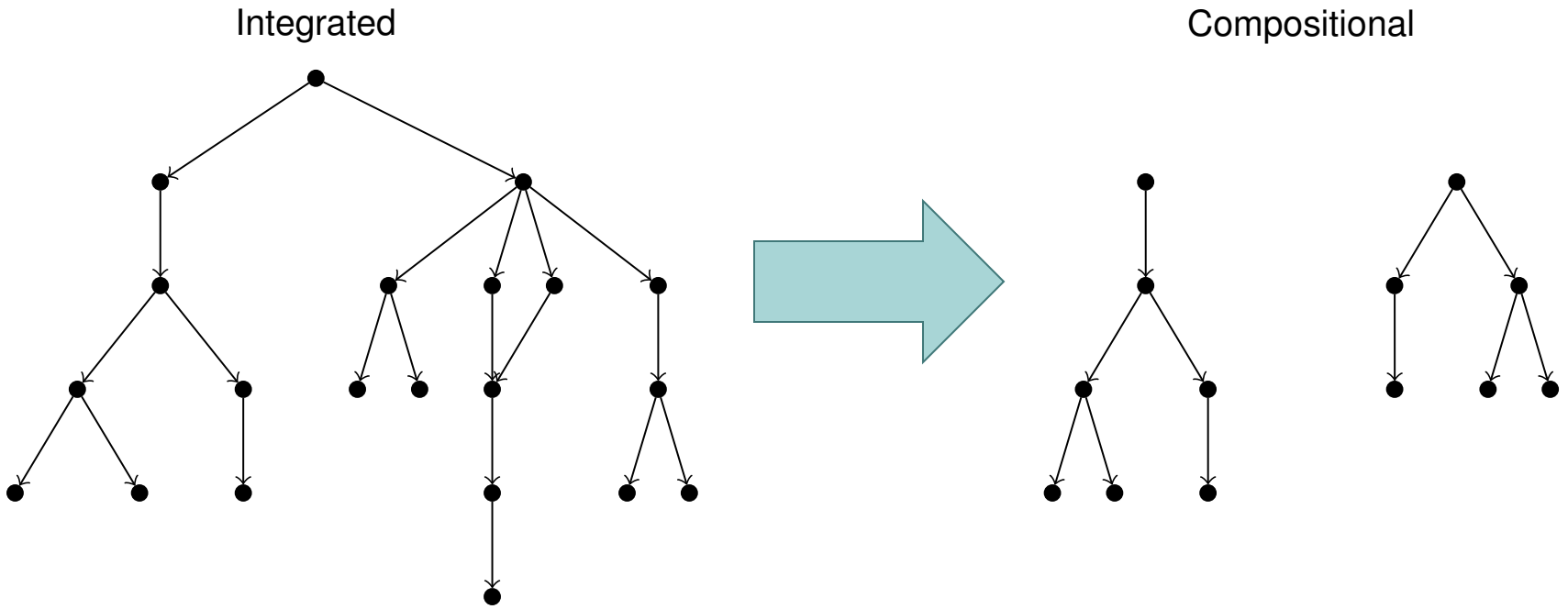
Timing Compositionality: By Example



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



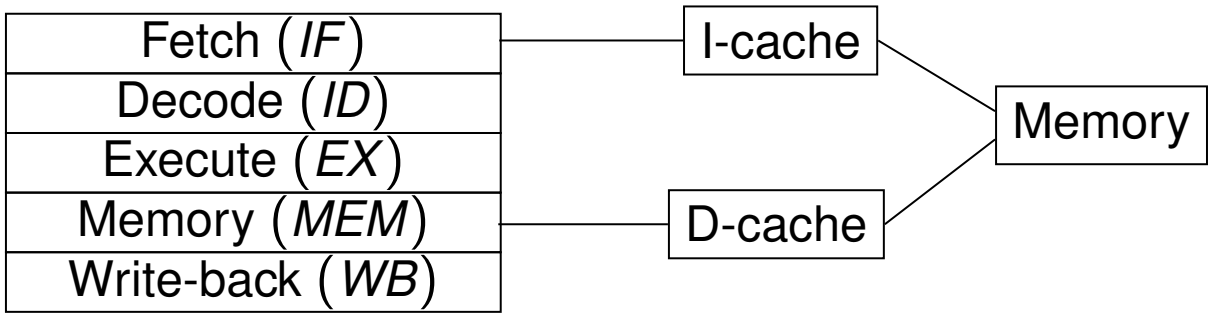
Conventional Wisdom

Simple in-order pipeline + LRU caches

→ no timing anomalies

→ timing-compositional

Bad News I: Timing Anomalies

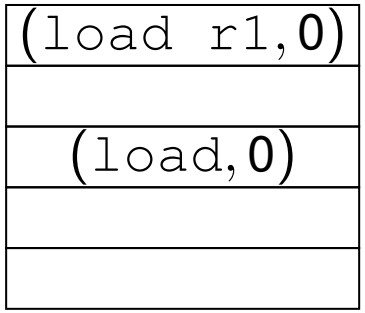


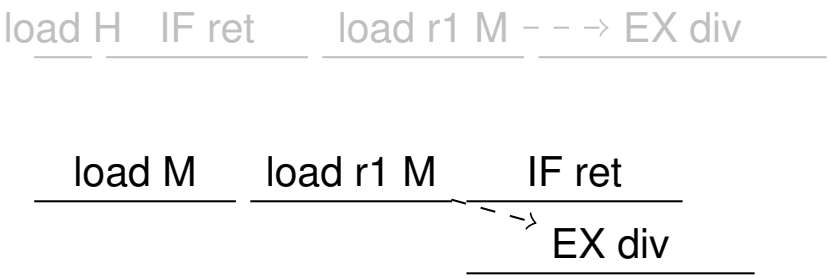
We show such a pipeline has timing anomalies:

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
```





Hit case:

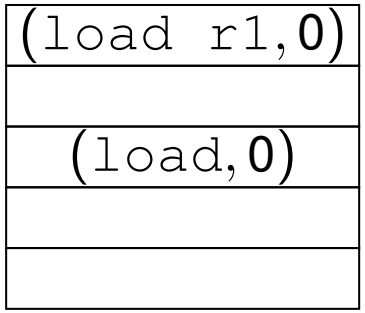
- Instruction fetch starts before second load becomes ready
- Stalls second load, which misses the cache

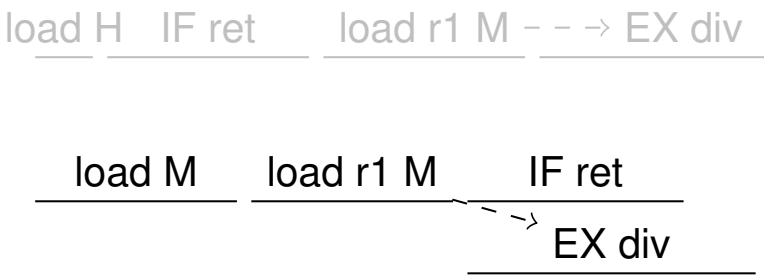
Miss case:

- Second load can catch up during first load missing the cache
- Second load is prioritized over instruction fetch
- Loading before fetching suits subsequent execution

A Timing Anomaly

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





Hit case:

- Instruction fetch starts before second load becomes ready
- S

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

Mis

- S
- 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

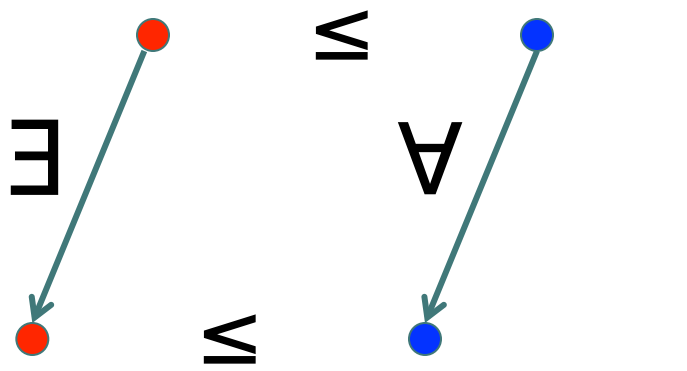
Definition (Strictly In-Order):
We call a pipeline *strictly in-order* if each *resource* processes the instructions in program order.

- 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

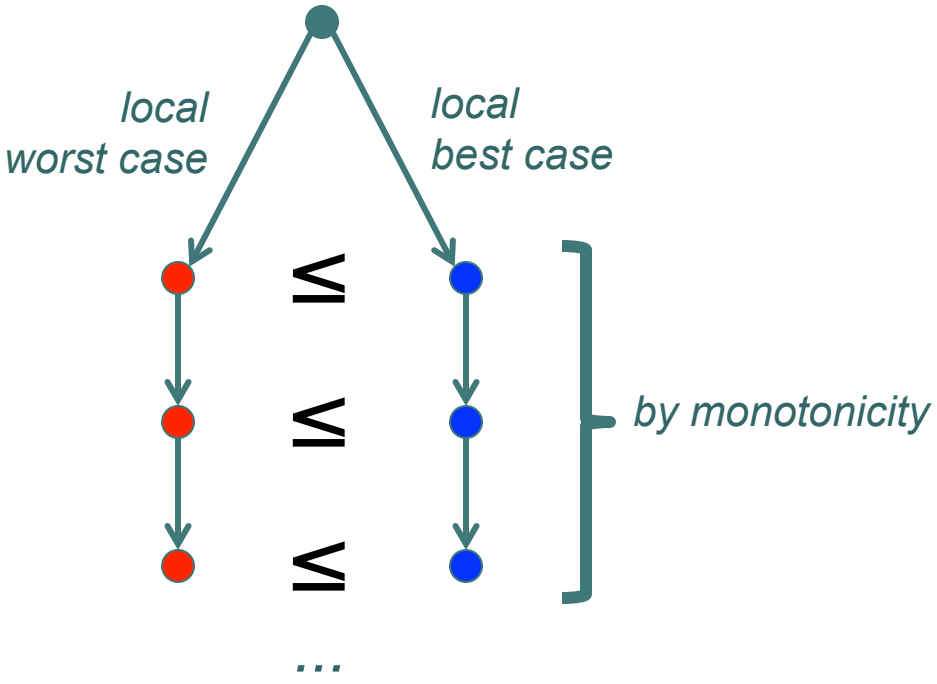
Theorem 1 (Monotonicity):
In the strictly in-order pipeline progress of an instruction is monotone in the progress of other instructions.

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



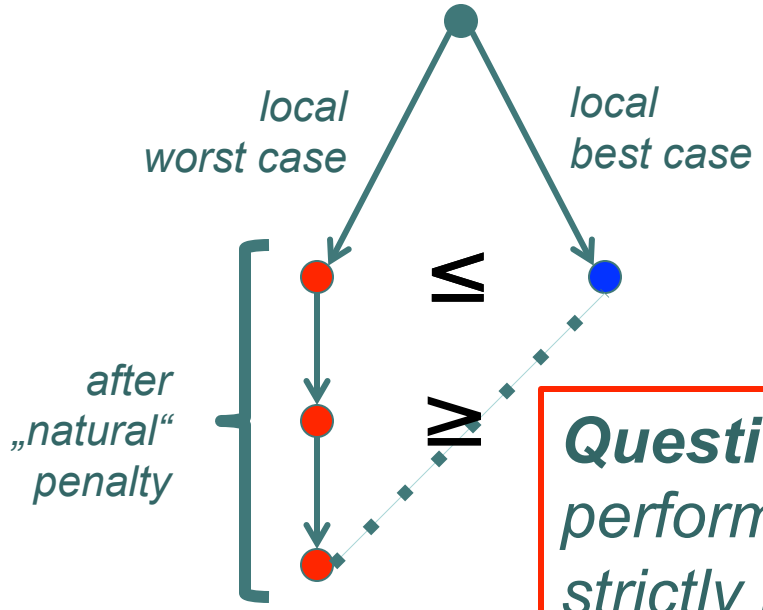
Strictly In-Order Pipelines: Properties

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.“



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

Strictly In-Order Pipelines: Experimental Evaluation

- On Mälardalen benchmarks:
  - 7% slowdown vs standard in-order pipeline
  - Still 3x speedup over no pipelining

→ Almost all the gains of pipelining but no anomalies and provably timing-compositional

Conclusions

- Need *faithful* timing models
- Various approaches to achieve predictability:
  - *Eliminate stateful components*
  - *Eliminate interference/temporal isolation*
  - *Choose „forgetful“ components*
  - Achieving **resource efficiency** is real challenge
- *Analyzability may in addition profit from*
  - *No Timing Anomalies*
  - *Timing Compositionality*

  } *Enable implicit treatment of large state spaces*

Case Studies: Predictable Microarchitectures

Academic Project at UC Berkeley:

Precision-Timed ARM (PTARM)

Goals: „Repeatable“ i.e. Deterministic Timing

Commercial Product by French Startup:

Kalray MPPA-256

Goals: No Timing Anomalies
+ Timing Compositionality

Precision-Timed ARM (PTARM)

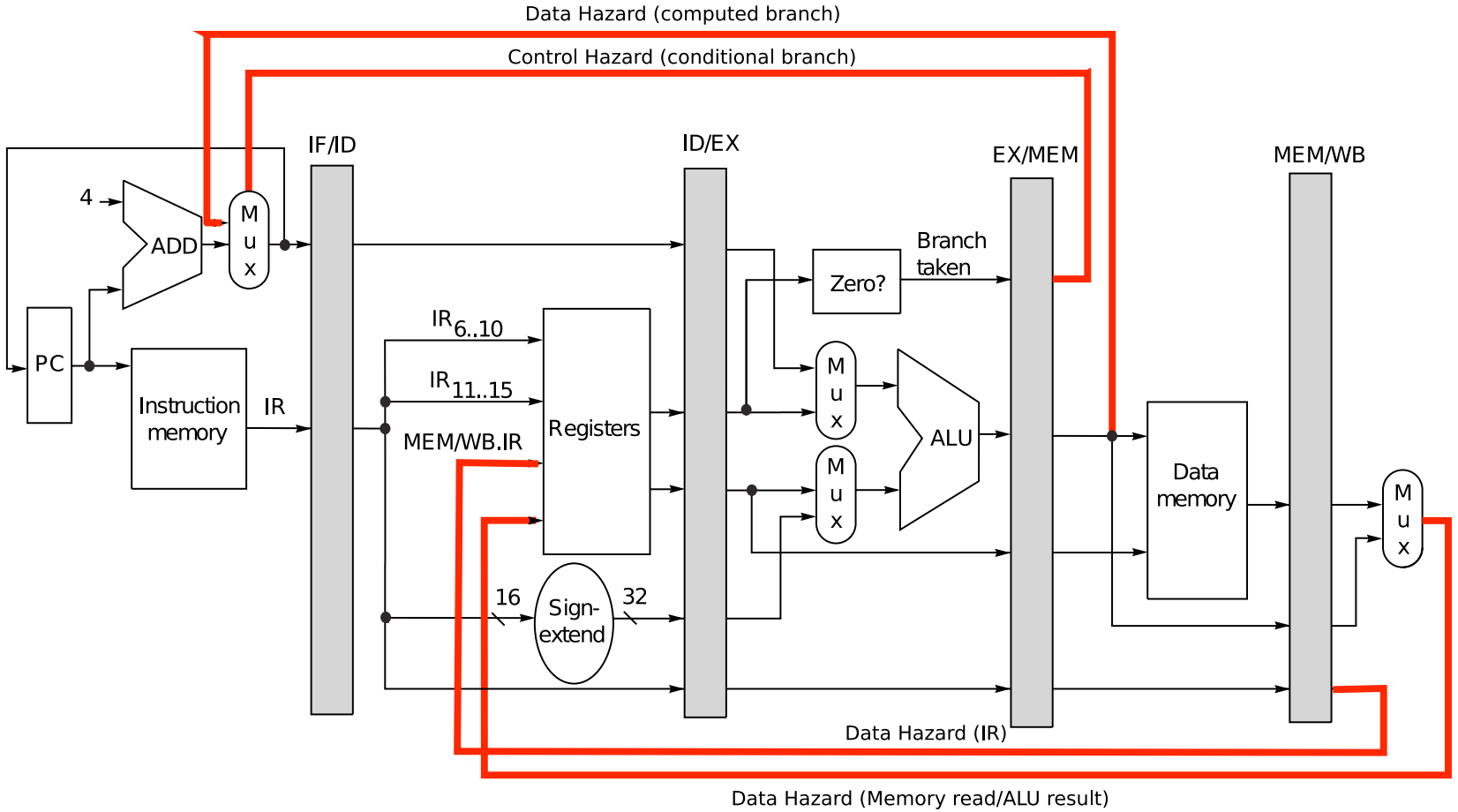
Initiated by Edward Lee (UC Berkeley) and Stephen Edwards (Columbia University) in 2007.

Main goal:

Make execution times completely deterministic.

For the same input, a program should always exhibit exactly the same execution time, independently of co-running tasks.

First Problem: Pipeline Hazards



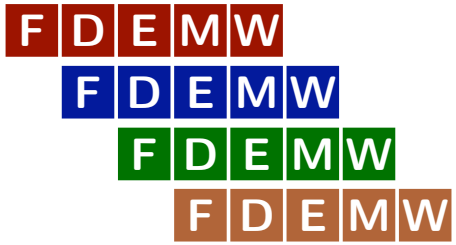
from Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 2007.

Forwarding helps, but not all the time…

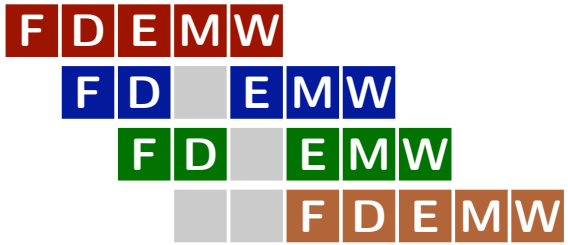
```
LD R1, 45(r2)
DADD R5, R1, R7
BE R5, R3, R0
ST R5, 48(R2)
```

Unpipelined

The Dream



The Reality



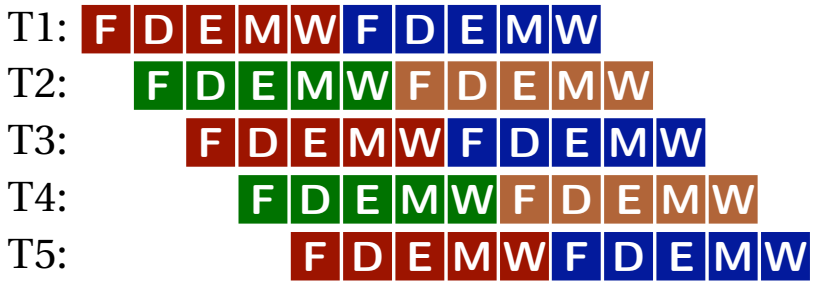
Memory Hazard

Data Hazard

Branch Hazard

Solution: Thread-interleaved Pipelines

+



Each thread occupies only one stage of the pipeline at a time

→ **No hazards**; perfect utilization of pipeline

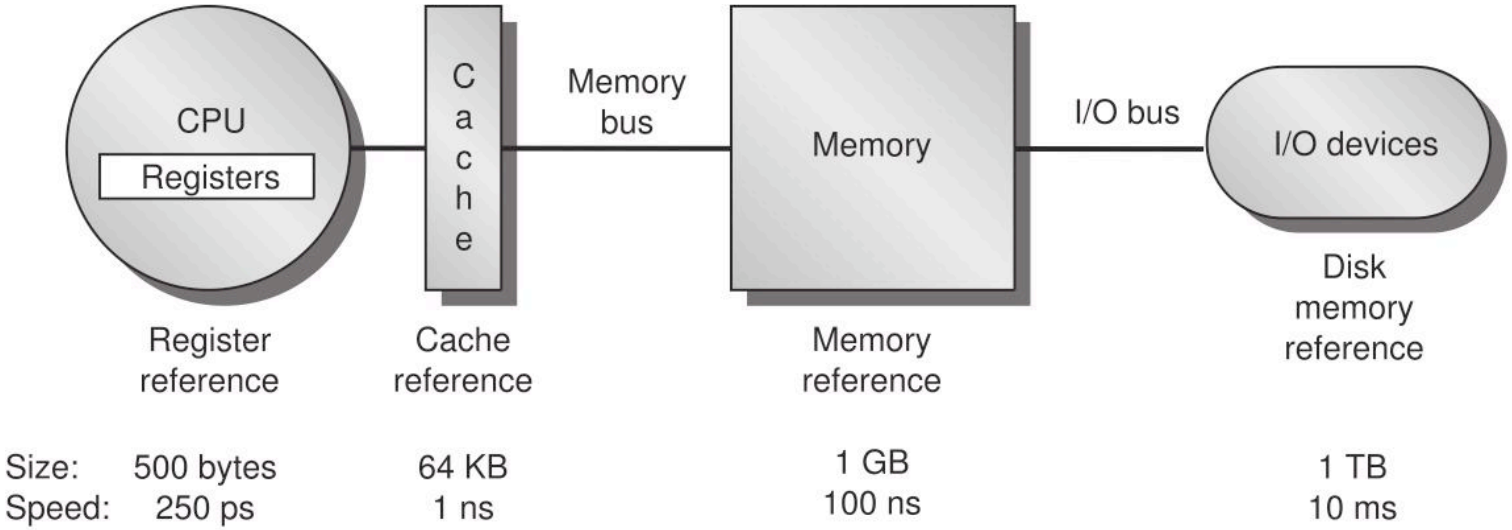
→ Simple hardware implementation (no forwarding, etc.)

→ Latency of instructions **independent** of micro-architectural state

→ Microarchitectural timing analysis becomes trivial

Drawback: reduced single-thread performance

Second Problem: Memory Hierarchy

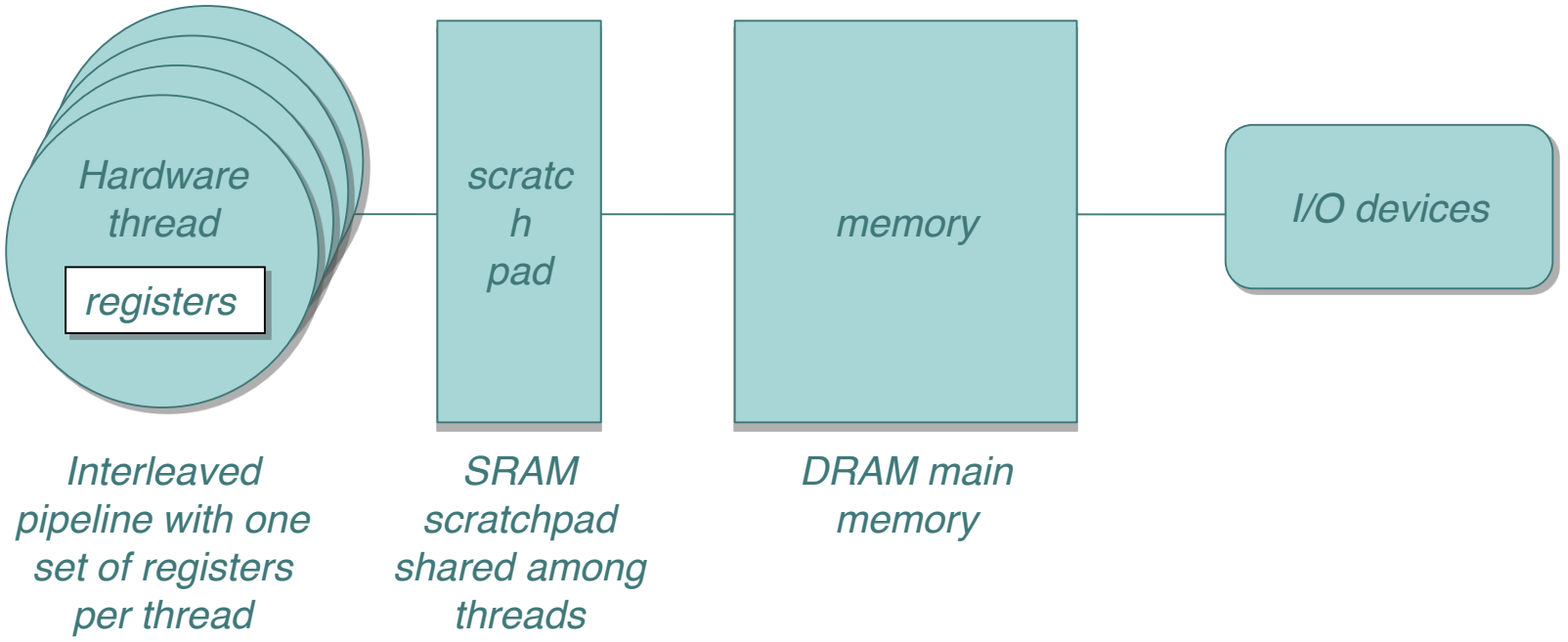


from Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 2007.

- Register file is a temporary memory under program control.
- Cache is a temporary memory under hardware control.

PRET principle: any temporary memory is under program control.

PRET principles implies Scratchpad in place of Cache

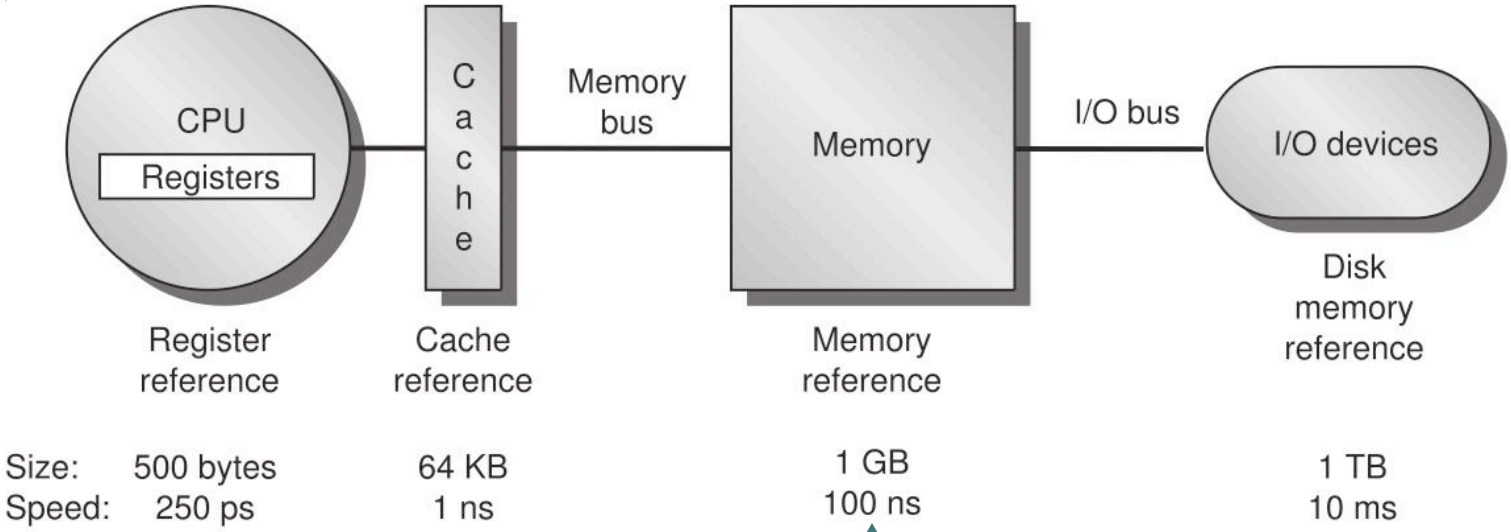


Scratchpad = *small but fast memory under software control*

Advantage: behavior perfectly predictable

Disadvantage: requires compiler support to manage contents

Second Problem: Memory Hierarchy



DRAM
- Slow ➔ High Latency
- High Capacity
- History-dependent access latency due to row buffer (= DRAM-internal cache)
- Accesses from different cores may interfere

from Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 2007.

Dynamic RAM Organization Overview

DRAM Cell

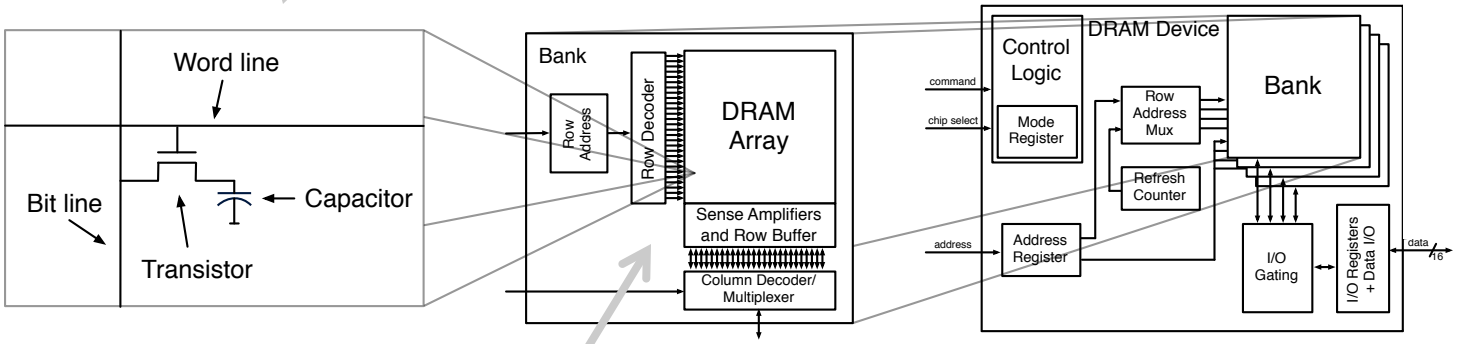
Leaks charge ➔ Needs to be refreshed (every 64ms for DDR2/DDR3) therefore "dynamic"

DRAM Device

Set of DRAM banks +

- *Control logic*
- *I/O gating*

***Accesses to banks can be pipelined**, however I/O + control logic are shared*



DRAM Bank

= Array of DRAM Cells + Sense Amplifiers and Row Buffer

Sharing of sense amplifiers and row buffer

General-Purpose DRAM Controller vs PRET DRAM Controller

General-Purpose Controller

- Abstracts DRAM as a single shared resource
- Schedules refreshes dynamically
- Schedules commands dynamically
- “Open page” policy speculates on locality

PRET DRAM Controller

- Exposes DRAM banks as independent resources
- Refreshes as reads: shorter interruptions
- Defer refreshes: improves perceived latency
- Follows periodic, time-triggered schedule
- “Closed page” policy: access-history independence

General-Purpose DRAM Controller vs PRET DRAM Controller

General-Purpose Controller

- Abstracts DRAM as a single shared resource
- Schedules refreshes dynamically
- Schedules commands dynamically
- "Open p… specula…

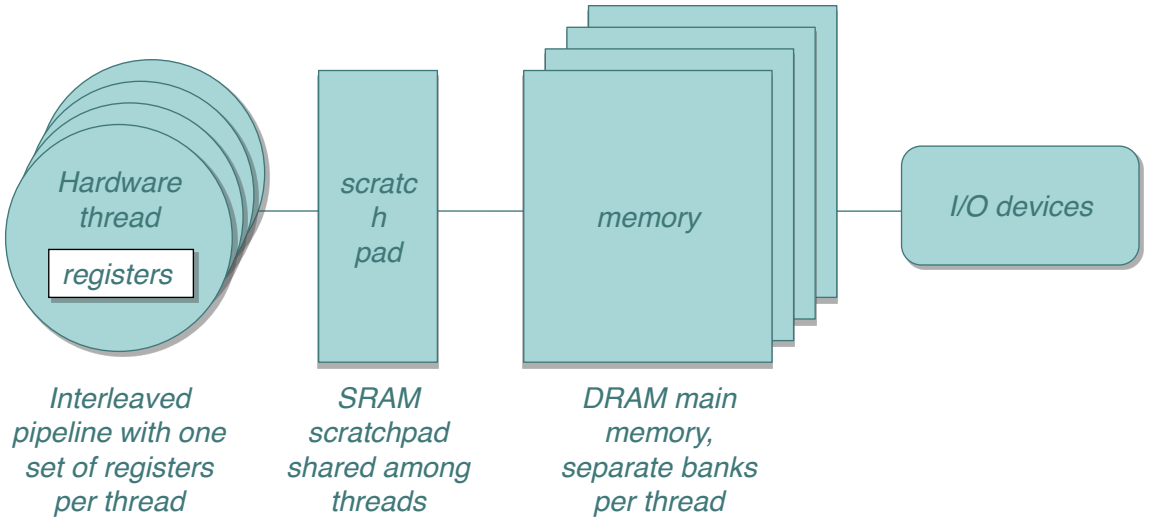
PRET DRAM Controller

- Exposes DRAM banks as independent resources
- Refreshes as reads: shorter interruptions
- Follows periodic, time-triggered schedule

Consequence:
→ low and constant worst-case access latencies!

See Reineke, Liu, Patel, Kim, Lee: PRET DRAM controller: Bank privatization for predictability and temporal isolation, in CODES+ISSS 2011 *for more details*.

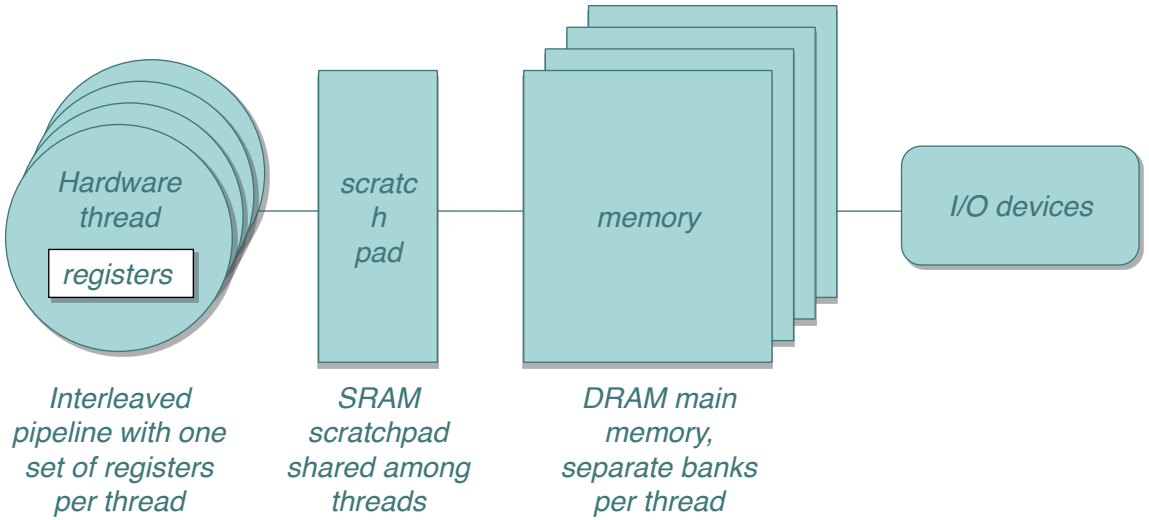
Precision-Timed ARM (PTARM)
Architecture Overview http://chess.eecs.berkeley.edu/pret/



- **Thread-Interleaved Pipeline** for predictable timing sacrificing latency but not throughput
- One private DRAM Resource + DMA Unit per Hardware Thread
- Shared **Scratchpad** Instruction and Data Memories for low-latency access

Precision-Timed ARM (PTARM)
Properties

http://chess.eecs.berkeley.edu/pret/



- Constant instruction latencies, independently of execution history or co-running tasks
- Reduced single-thread throughput: similar to non-pipelined processor
- Aggregate (all hardware threads combined) throughput higher than standard in-order pipeline

Kalray MPPA

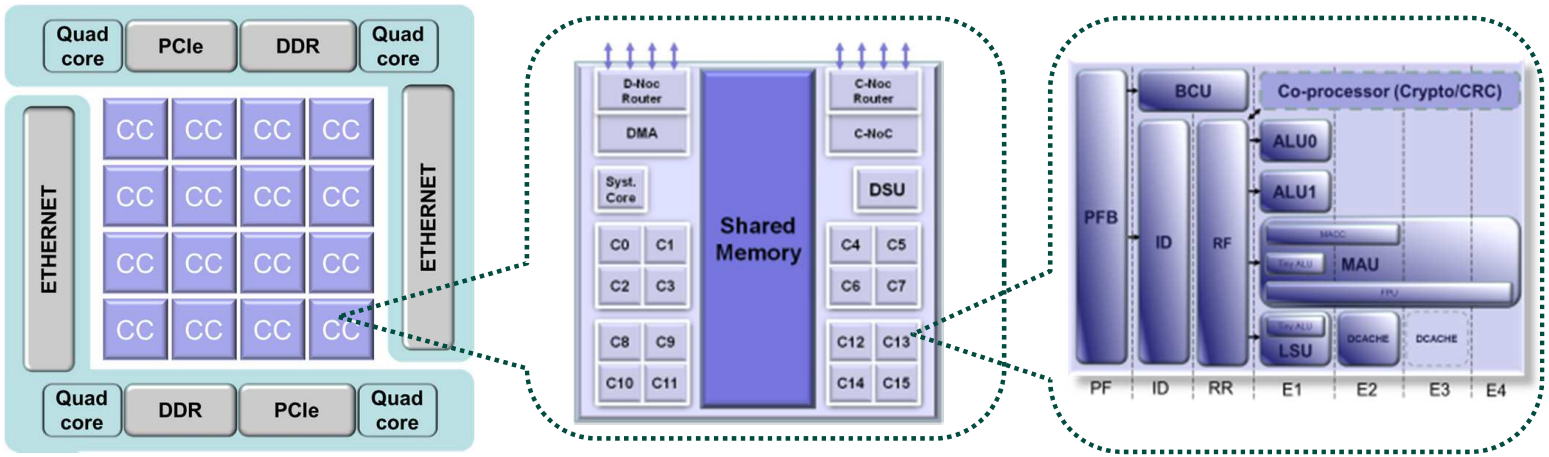
Kalray = French semiconductor company
founded in 1999

MPPA = Massively Parallel Processor Array

Main goal:

Achieve timing predictability by ensuring timing compositionality and enabling temporal isolation.

Kalray MPPA – High-level Structure



Many-core processor

- *16 compute clusters*
- *2 I/O clusters*
- *Data and control NoC with bandwidth guarantees*
- *Distributed memory architecture*
- *634 GFLOPS*

Compute cluster

- *16 user cores + 1 system core*
- *2 MB multi-banked shared memory*
- *77 GB/s shared memory bandwidth*

VLIW core

- 5-issue (in-order) VLIW architecture
- I&D Caches (8KB + 8 KB), LRU replacement

Can be analyzed using network calculus techniques.

Can be configured to ensure temporal isolation.

Claimed to be timing compositional
Hard to confirm or refute. Doubtful given our results about standard in-order pipelines.

Figure copyright: Kalray SA

Kalray MPPA – Compute Cluster Structure

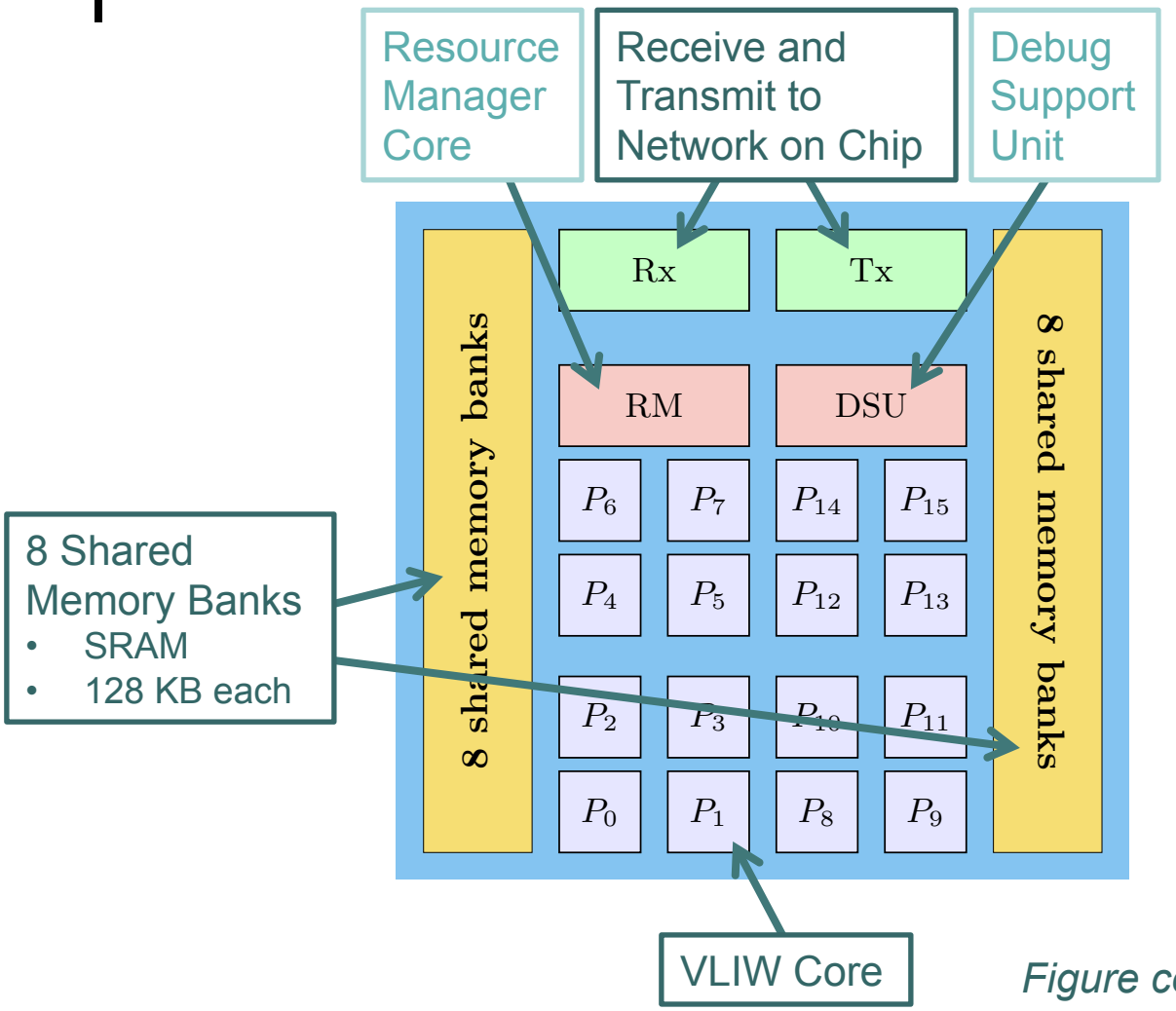
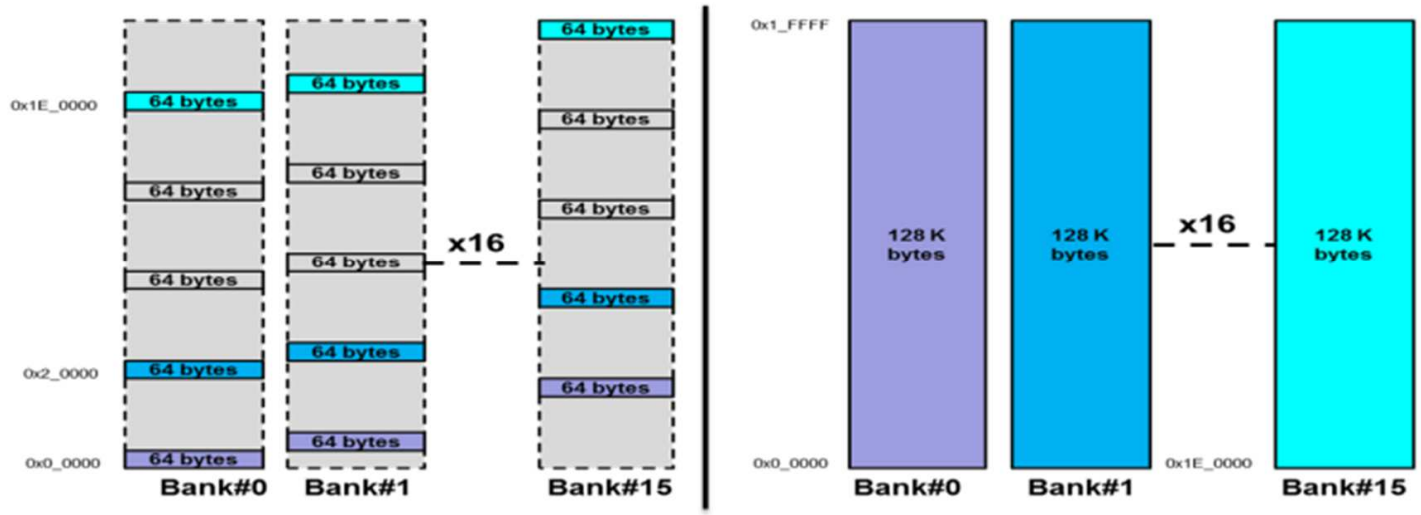


Figure courtesy of Hamza Rihani

Kalray MPPA
Access Modes to Memory Banks

1. Interleaved:

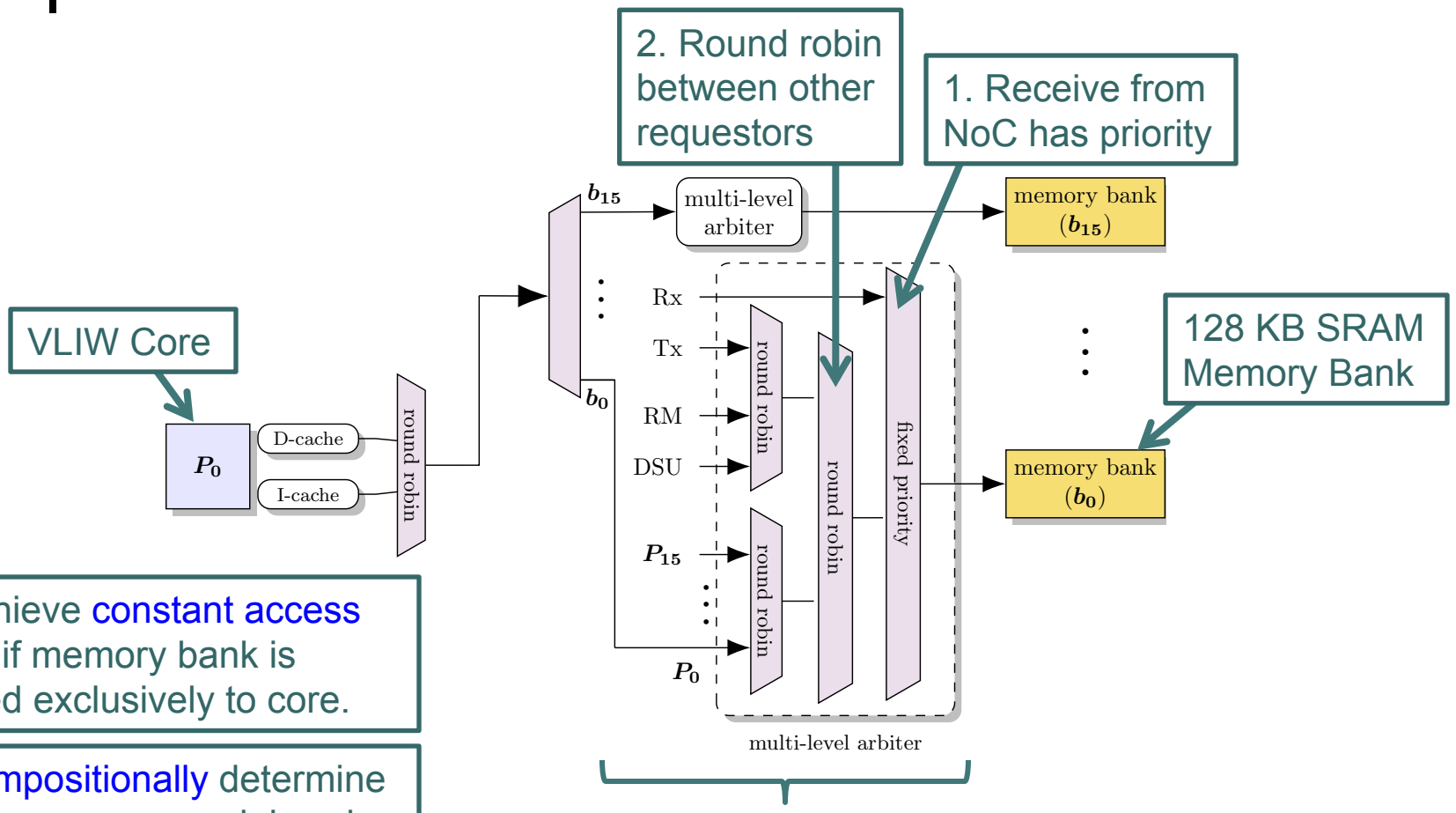
2. Banked:



Banked mode may be used to allocate private banks to applications on different cores for temporal isolation.

Figure copyright: Kalray SA

Kalray MPPA
Accessing Memory Banks from Cores



Can achieve constant access latency if memory bank is assigned exclusively to core.

Can compositionally determine total memory access delays in case of interference.

Separate Interconnect and Arbitration for each Memory Bank

Figure courtesy of Hamza Rihani

Kalray MPPA

- Clusters can be configured to achieve temporal isolation between tasks on different cores
- Interference on intra-cluster network can be analyzed compositionally
- In-order VLIW cores with LRU caches for predictability
  - Timing compositionality somewhat doubtful

Literature:
Timing Anomalies/Timing Compositionality

- Lundqvist, Stenström: Timing anomalies in dynamically scheduled microprocessors, In: Proceedings RTSS, 1999
- Reineke et al.: A definition and classification of timing anomalies, In: Proceedings WCET, 2006
- Hahn, Reineke, Wilhelm: Towards compositionality in execution time analysis – definition and challenges. In: Proceedings CRTS, 2013
- Hahn, Reineke, Wilhelm: Toward Compact Abstractions for Processor Pipelines, In: Correct System Design, 2015.
- Hahn, Jacobs, Reineke: Enabling compositionality for multicore timing analysis, In: Proceedings RTNS, 2016

Literature: Timing Predictability

- Thiele, Wilhelm: Design for timing predictability, Real-Time Systems, 2004
- Reineke, Grund, Berg, Wilhelm: Timing predictability of cache replacement policies, Real-Time Systems 37(2), 2007
- Wilhelm et al.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems, IEEE TCAD, 2009
- Bui, Lee, Liu, Patel, Reineke: Temporal isolation on multiprocessing architectures, In: Proceedings DAC, 2011
- Axer et al.: Building timing predictable embedded systems, ACM TECS 13(4), 2014
- Reineke, Salinger: On the Smoothness of Paging Algorithms, In: Proceedings WAOA, 2015

Literature:
Case Studies: Kalray MPPA, PRET

- Dupont de Dinechin, van Amstel, Poulhies, Lager: Time-critical computing on a single-chip massively parallel processor, In: Proceedings DATE, 2014
- Edwards, Lee: The case for the precision timed (PRET) machine, In: DAC, 2007
- Reineke, Liu, Patel, Kim, Lee: PRET DRAM controller: Bank privatization for predictability and temporal isolation, In: Proceedings CODES+ISSS, 2011
- Liu, Reineke, Broman, Zimmer, Lee: A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance, In: Proceedings ICCD, 2012

Literature: Modeling Microarchitectures

- Abel, Reineke: Measurement-based Modeling of the Cache Replacement Policy In: Proceedings RTAS, 2013.
- Abel, Reineke: Reverse engineering of cache replacement policies in intel microprocessors and their evaluation, In: Proceedings ISPASS, 2014
- Hassan, Kaushik, Patel: Reverse-engineering embedded memory controllers through latency-based analysis, In: Proceedings RTAS, 2015