Challenges for Worst-case Execution Time Analysis of Multi-core Architectures

Jan Reineke @ SAARLAND UNIVERSITY
COMPUTER SCIENCE

Intel, Braunschweig

April 29, 2013

The Context: Hard Real-Time Systems

Safety-critical applications:

- Avionics, automotive, train industries, manufacturing

Side airbag in car
Reaction in < 10 msec

Crankshaft-synchronous tasks
Reaction in < 45 microsec

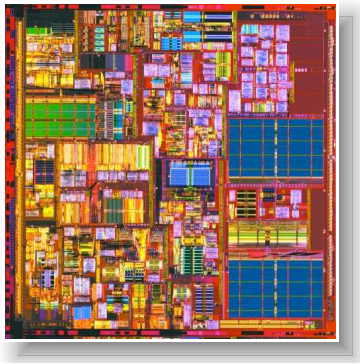
- Embedded controllers must finish their tasks within given time bounds.
- Developers would like to know the Worst-Case Execution Time (WCET) to give a guarantee.

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]);
}
```

Embedded Software

+



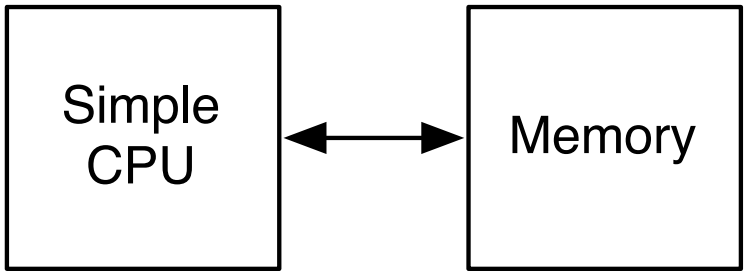
Microarchitecture

?

Timing Requirements

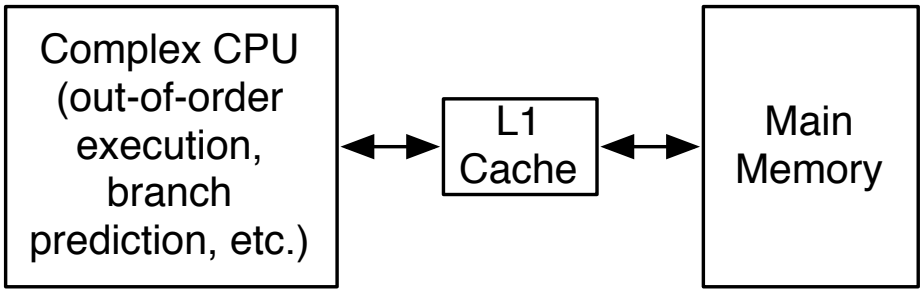
What does the execution time depend on?

- 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.



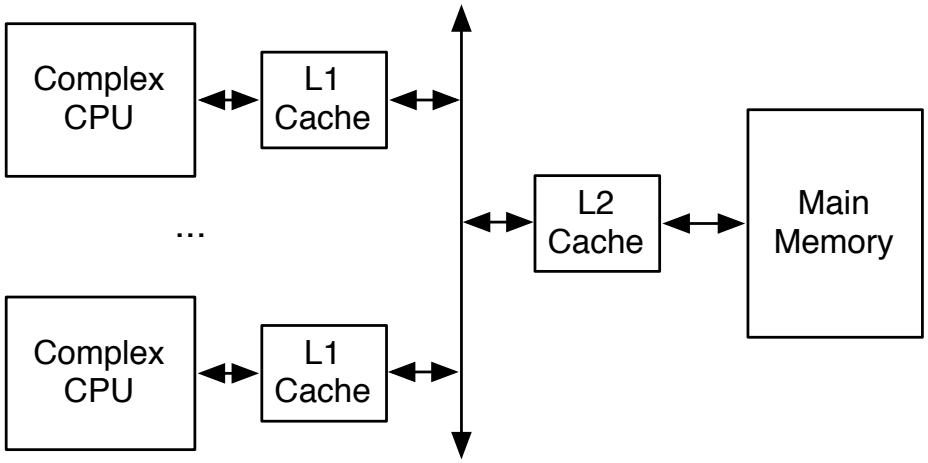
What does the execution time depend on?

- 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.



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.

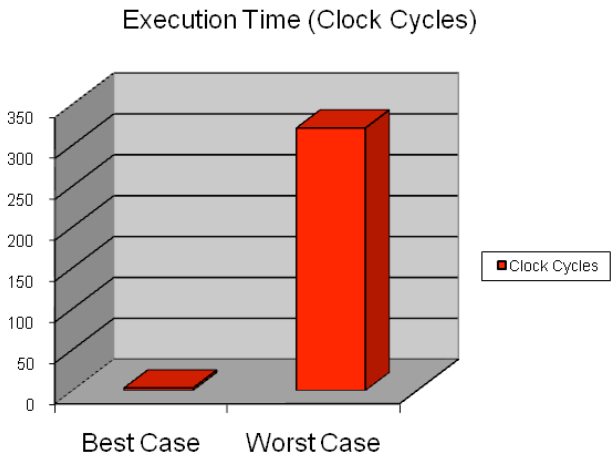


Example of Influence of Microarchitectural State

x=a+b; →

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

PowerPC 755



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

Challenges

1. Modeling

   How to construct sound timing models?

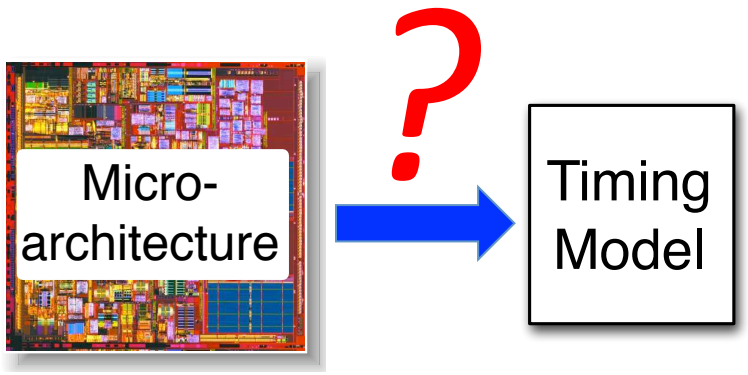
2. Analysis

   How to precisely & efficiently bound the WCET?

3. Design

   How to design microarchitectures that enable precise & efficient WCET analysis?

The Modeling Challenge

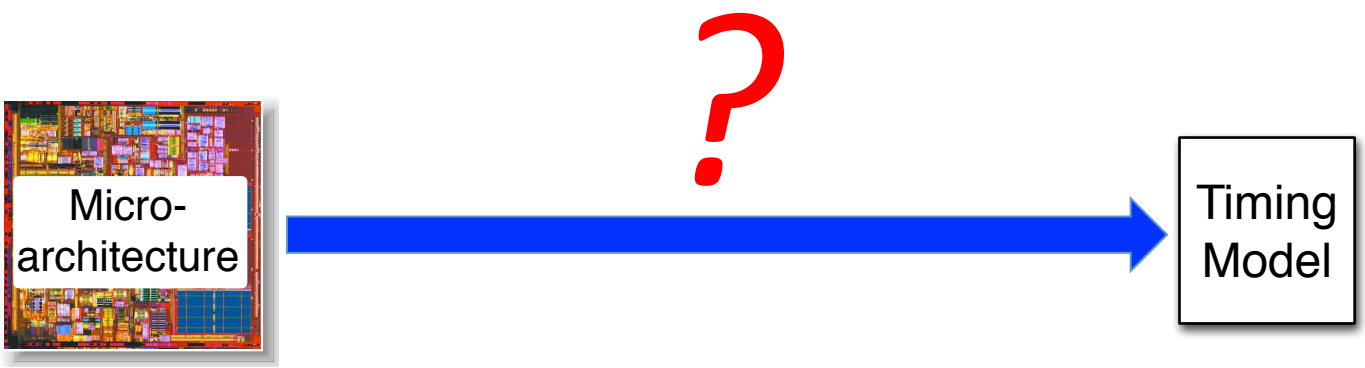


Timing model = Formal specification of microarchitecture's timing

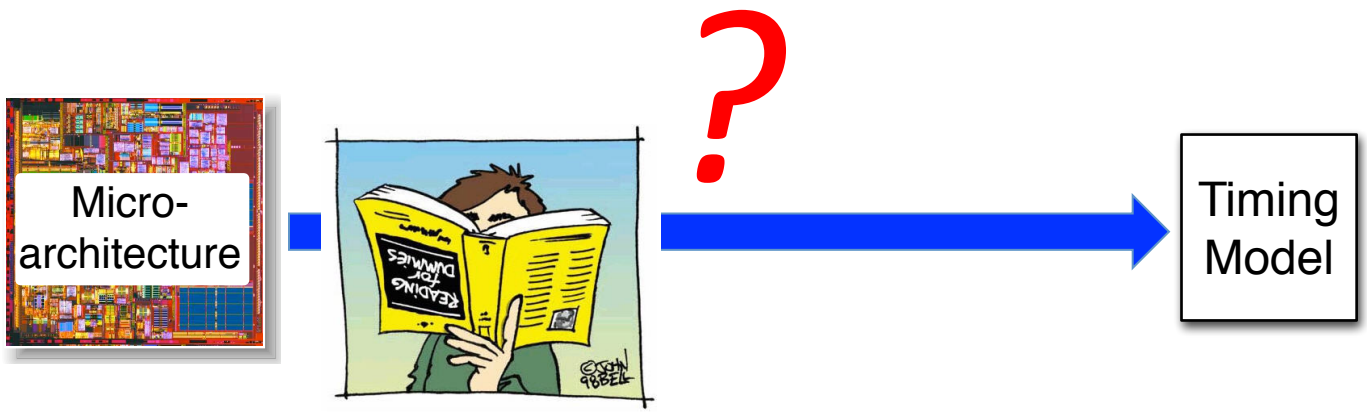
Incorrect timing model

→ possibly incorrect WCET bound.

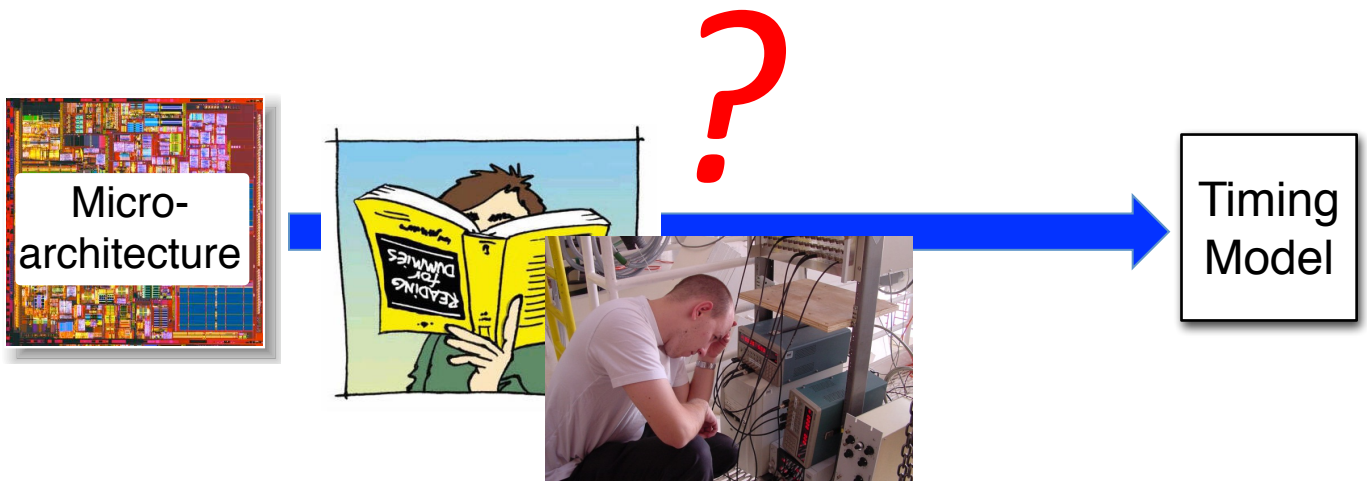
Current Process of Deriving Timing Model



Current Process of Deriving Timing Model



Current Process of Deriving Timing Model



Current Process of Deriving Timing Model

Current Process of Deriving Timing Model

→ Time-consuming, and

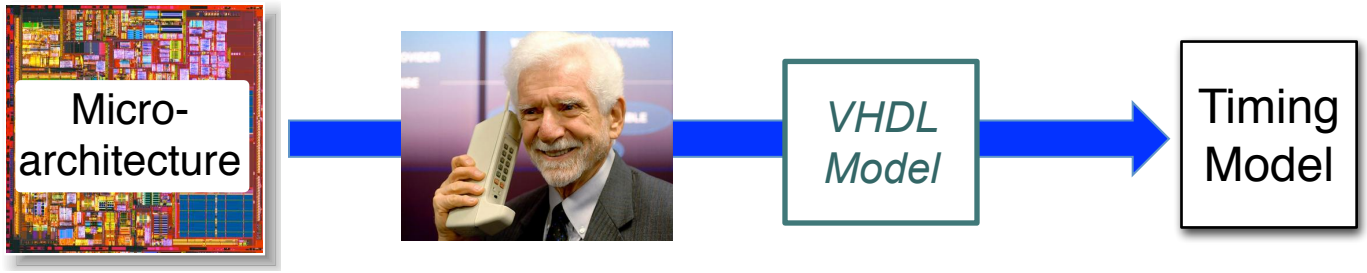
→ error-prone.

Current Process of Deriving Timing Model

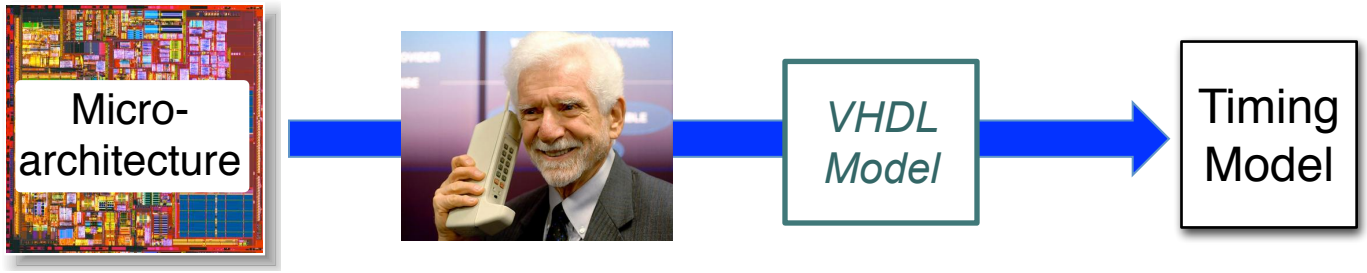
→ Time-consuming, and

→ error-prone.

1. Future Process of Deriving Timing Model



1. Future Process of Deriving Timing Model

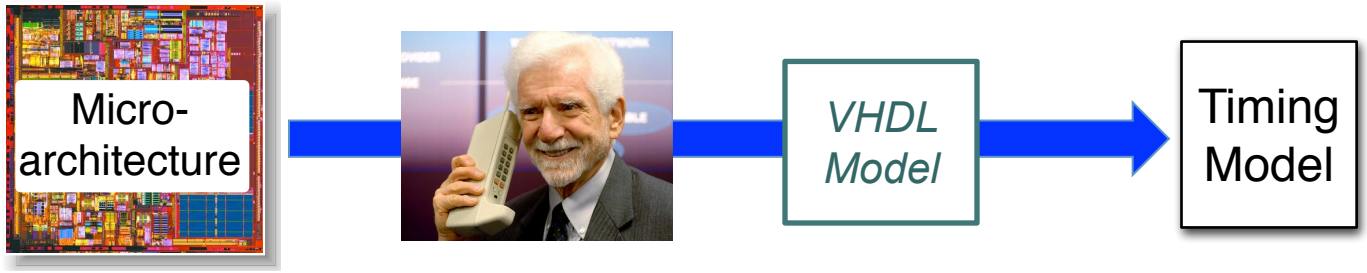


Derive timing model automatically from formal specification of microarchitecture.

→ *Less manual effort, thus less time-consuming, and*

→ *provably correct.*

1. Future Process of Deriving Timing Model

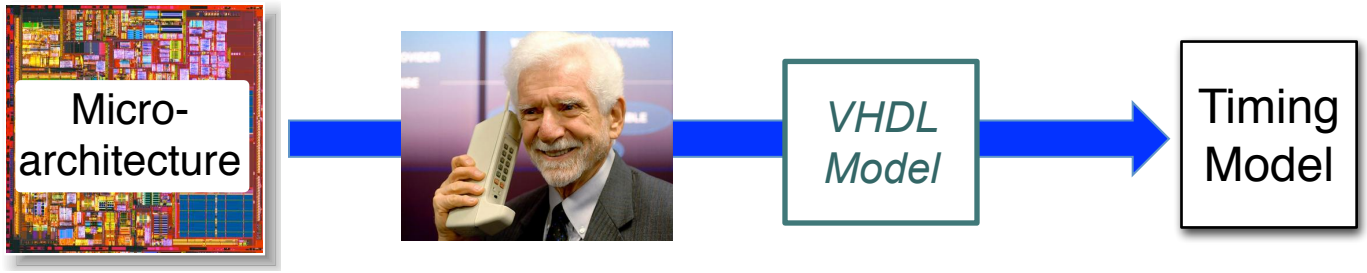


Derive timing model automatically from formal specification of microarchitecture.

→ *Less manual effort, thus less time-consuming, and*

→ *provably correct.*

1. Future Process of Deriving Timing Model

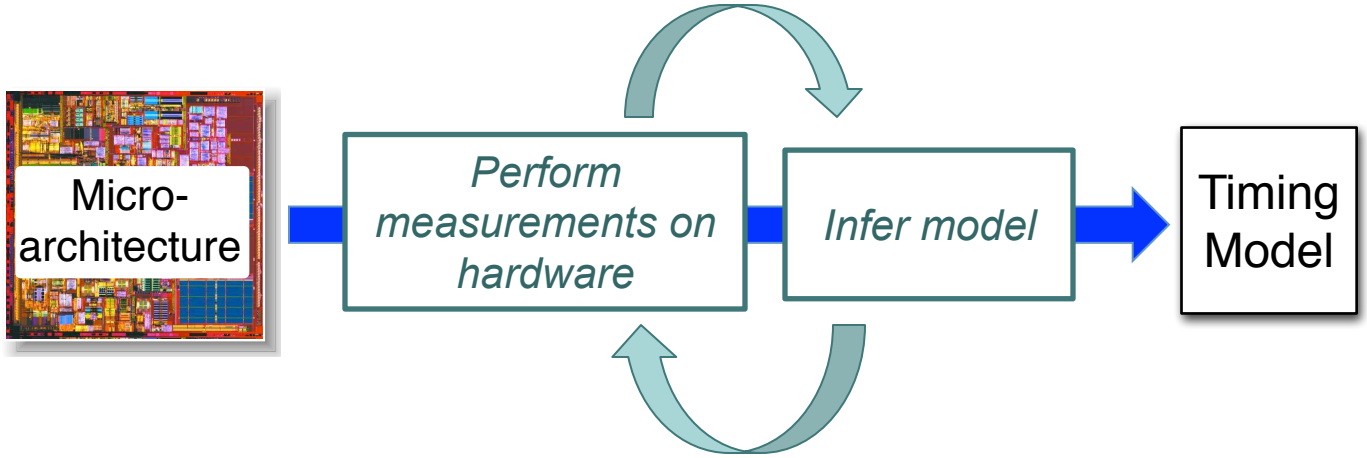


Derive timing model automatically from formal specification of microarchitecture.

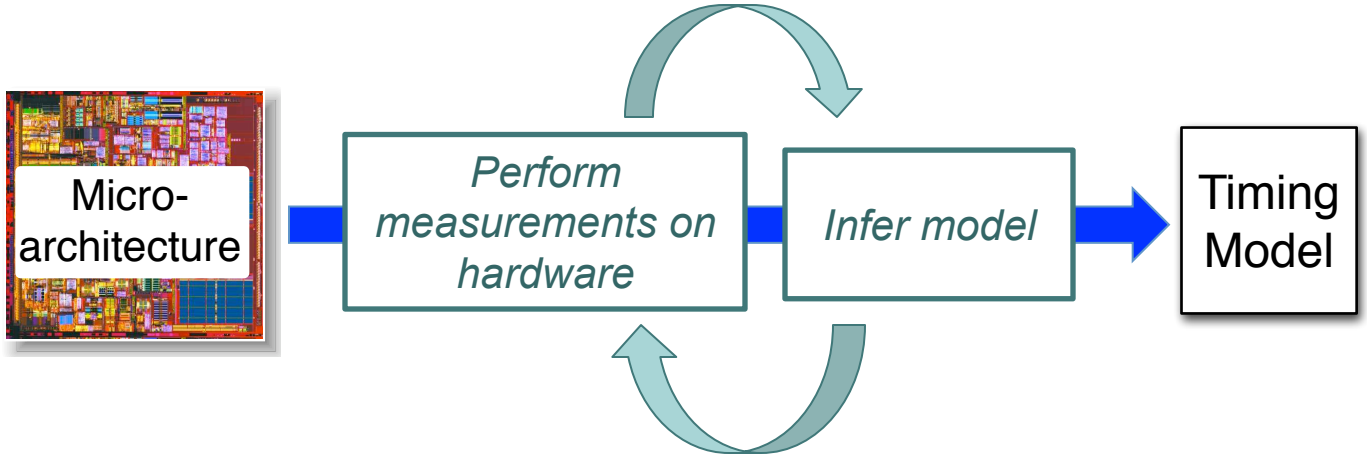
→ *Less manual effort, thus less time-consuming, and*

→ *provably correct.*

2. Future Process of Deriving Timing Model



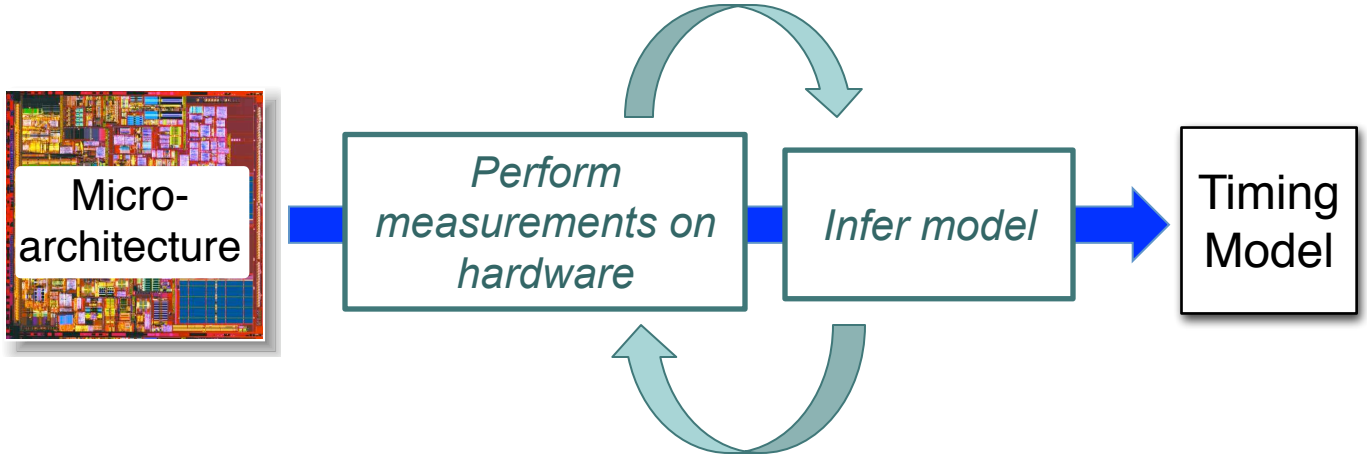
2. Future Process of Deriving Timing Model



Derive timing model automatically from measurements on the hardware using ideas from automata learning.

- → *No manual effort, and*
- → *(under certain assumptions) provably correct.*
- → *Also useful to validate assumptions about microarch.*

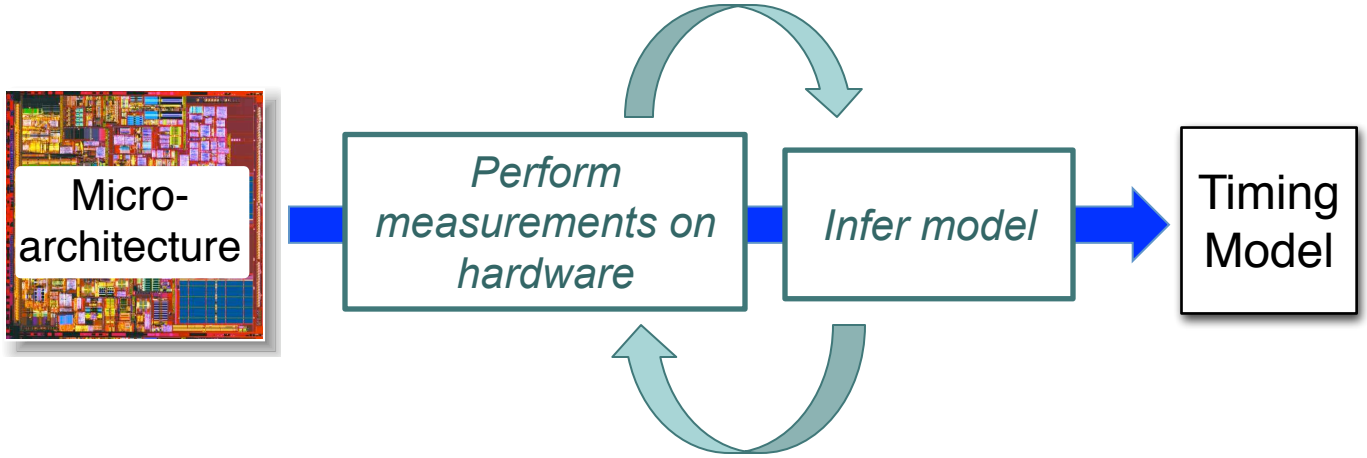
2. Future Process of Deriving Timing Model



Derive timing model automatically from measurements on the hardware using ideas from automata learning.

- → *No manual effort, and*
- → *(under certain assumptions) provably correct.*
- → *Also useful to validate assumptions about microarch.*

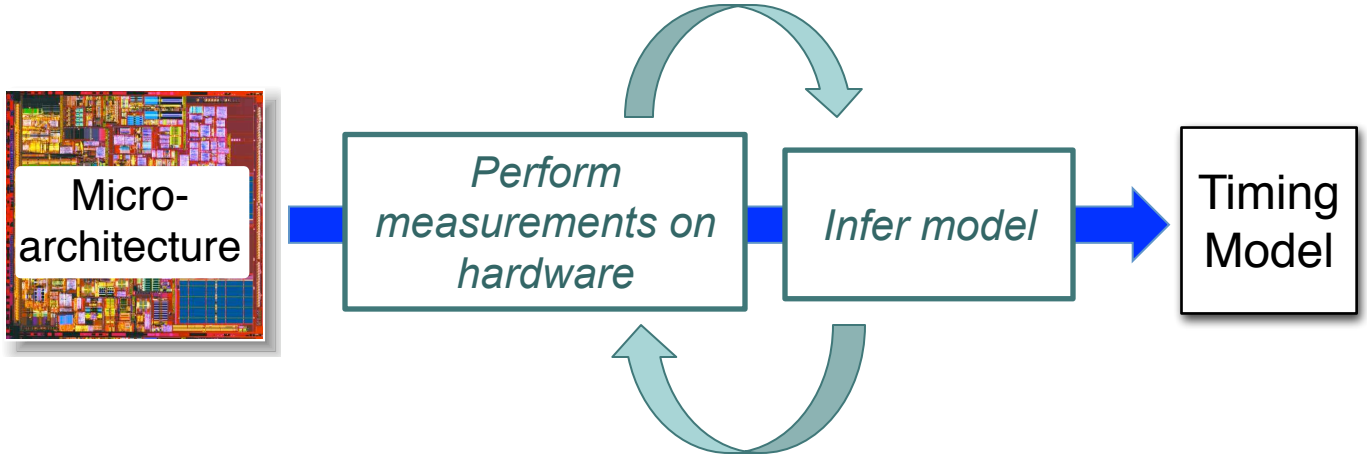
2. Future Process of Deriving Timing Model



Derive timing model automatically from measurements on the hardware using ideas from automata learning.

- → *No manual effort, and*
- → *(under certain assumptions) provably correct.*
- → *Also useful to validate assumptions about microarch.*

2. Future Process of Deriving Timing Model

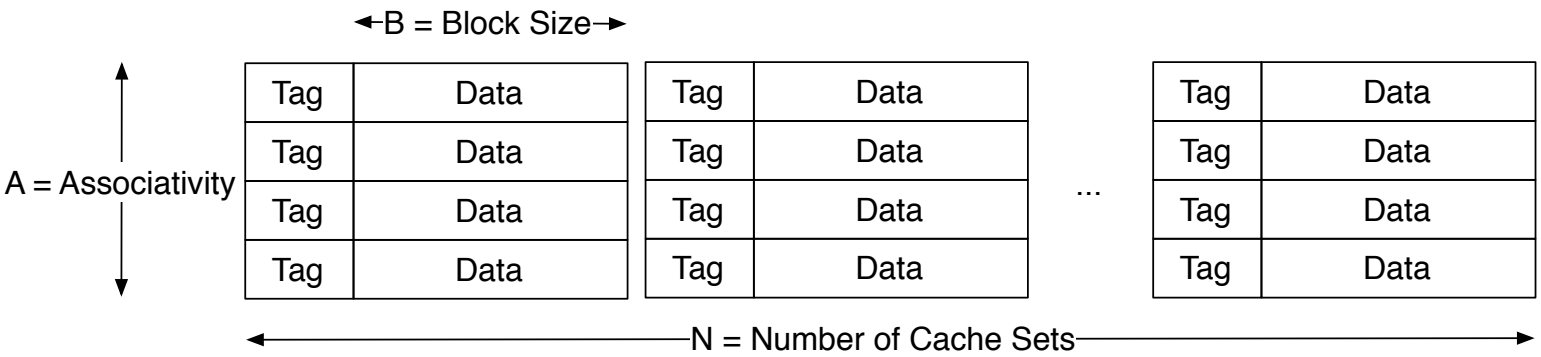


Derive timing model automatically from measurements on the hardware using ideas from automata learning.

- → *No manual effort, and*
- → *(under certain assumptions) provably correct.*
- → *Also useful to validate assumptions about microarch.*

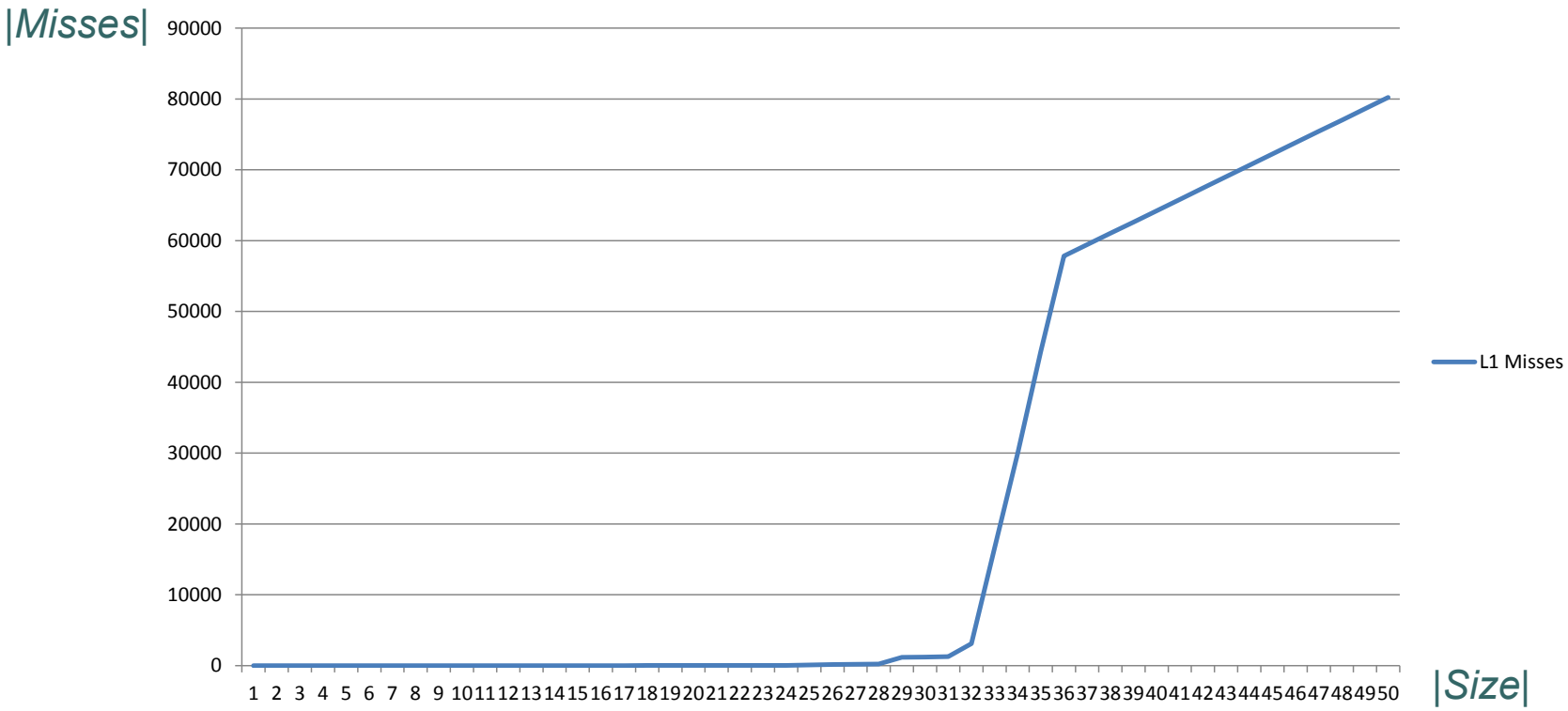
Proof-of-concept: Automatic Modeling of the Cache Hierarchy

- Cache Model is important part of Timing Model
- Can be characterized by a few parameters:
  - ABC: associativity, block size, capacity
  - Replacement policy

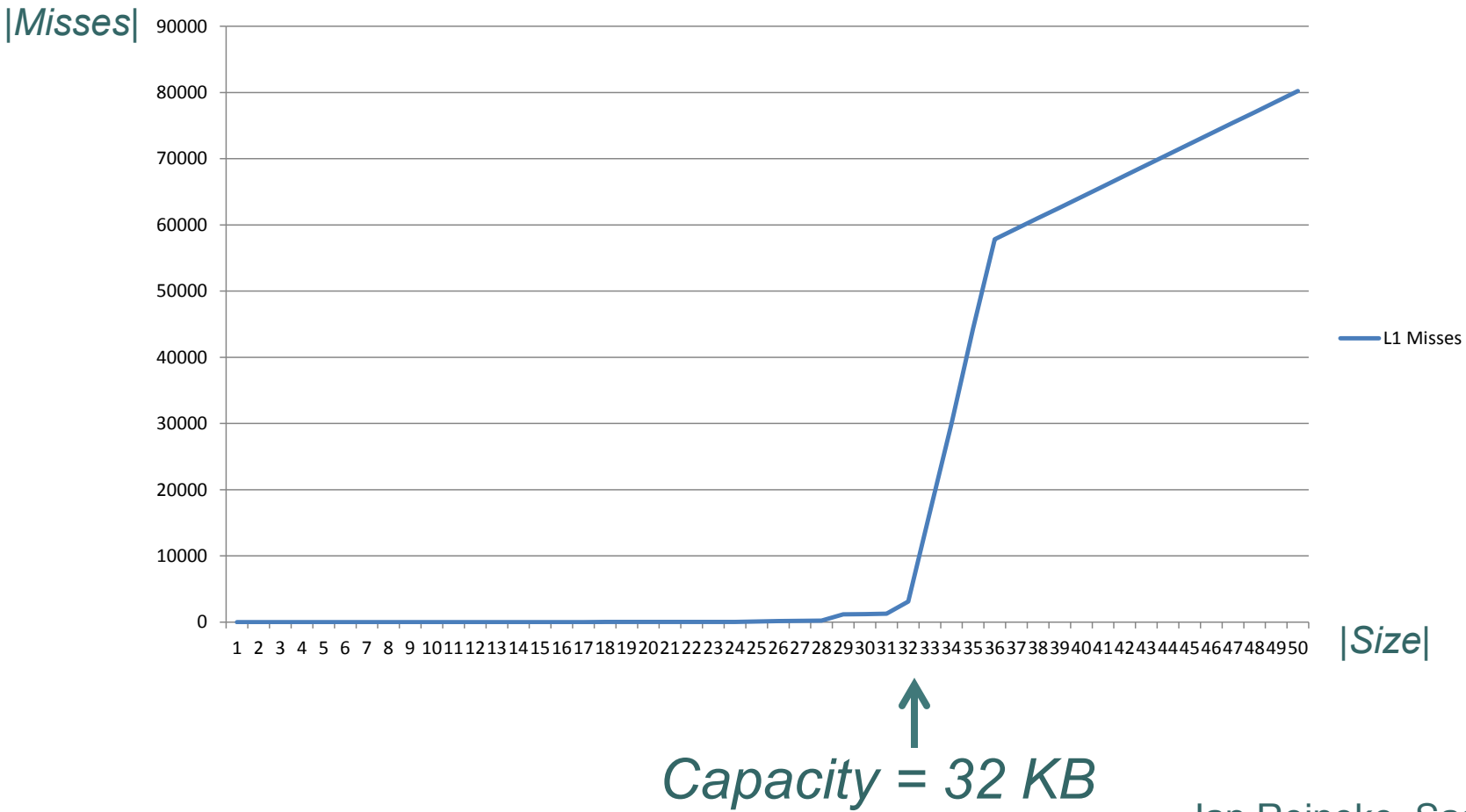


chi [Abel and Reineke, RTAS 2013] derives all of these parameters fully automatically.

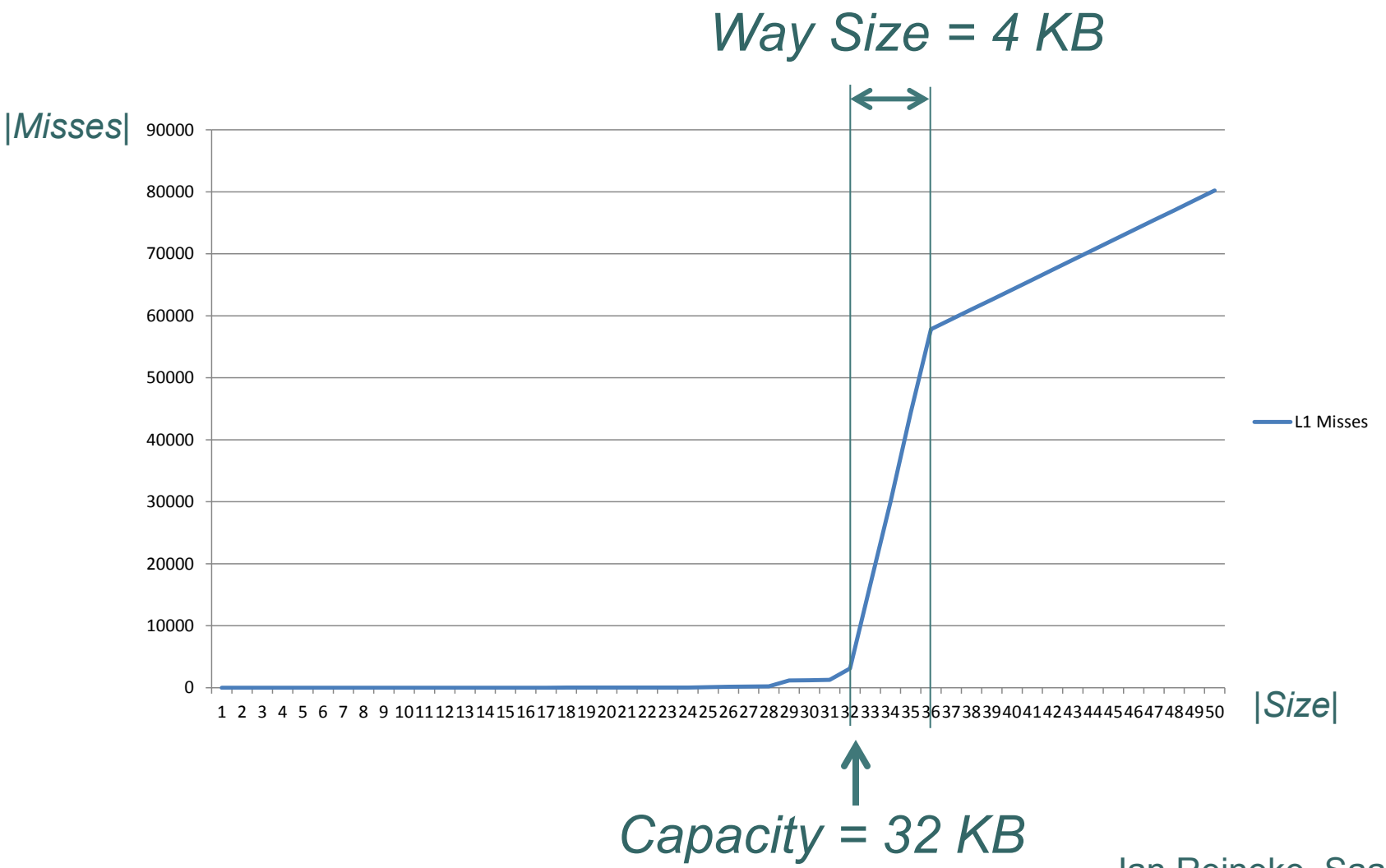
Example: Intel Core 2 Duo E6750, L1 Data Cache



Example: Intel Core 2 Duo E6750, L1 Data Cache



Example: Intel Core 2 Duo E6750, L1 Data Cache



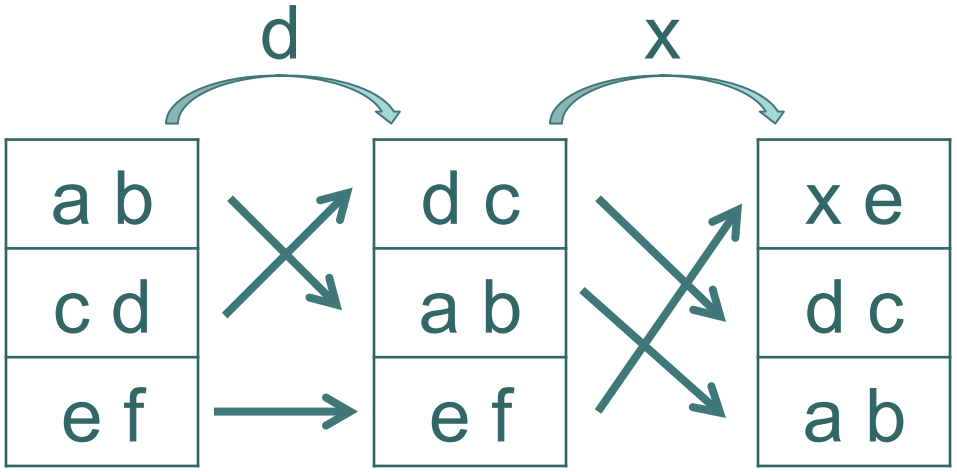
Replacement Policy

Approach inspired by methods to learn finite automata. Heavily specialized to problem domain.

Replacement Policy

Approach inspired by methods to learn finite automata. Heavily specialized to problem domain.

Discovered to our knowledge undocumented policy of the Intel Atom D525:



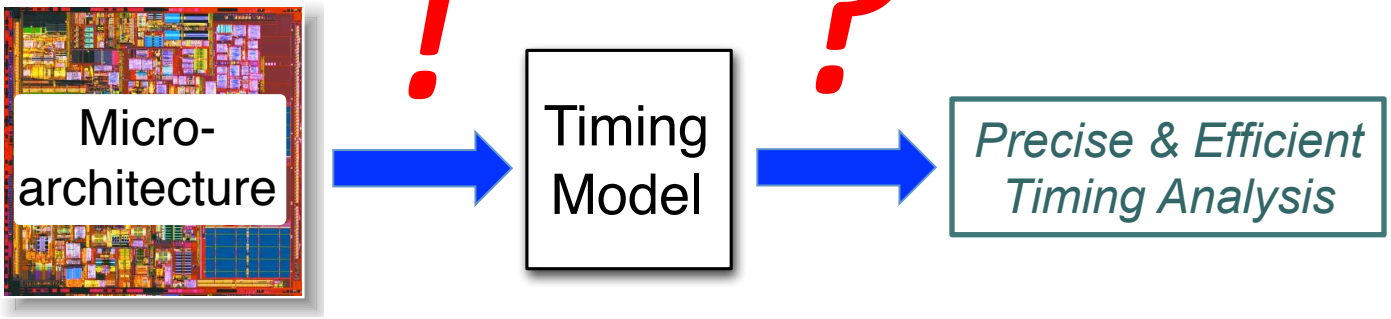
More information: http://embedded.cs.uni-saarland.de/chi.php

Modeling Challenge: Future Work

Extend automation to other parts of the microarchitecture:

- Translation lookaside buffers, branch predictors
- Shared caches in multicores including their coherency protocols
- Out-of-order pipelines?

The Analysis Challenge



Consider all possible program inputs

Consider all possible initial states of the hardware

$$WCET_H(P) := \max_{i \in Inputs} \max_{h \in States(H)} ET_H(P, i, h)$$

The Analysis Challenge

Consider all possible program inputs

Consider all possible initial states of the hardware

$$WCET_H(P) := \max_{i \in Inputs} \max_{h \in States(H)} ET_H(P, i, h)$$

Explicitly evaluating ET for all inputs and all hardware states is not feasible in practice:

- *There are simply too many.*

➔ *Need for abstraction and thus approximation!*

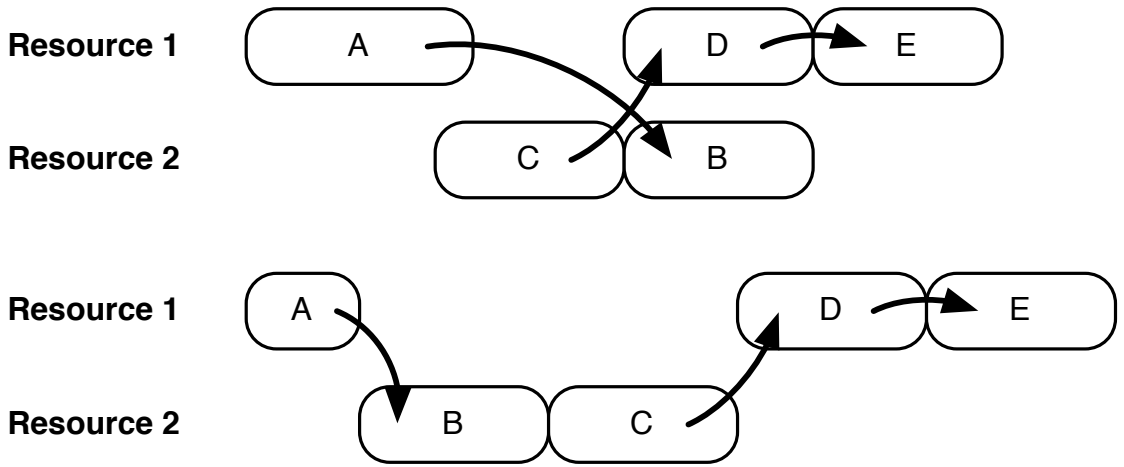
The Analysis Challenge: State of the Art

Component	Analysis Status
Caches, Branch Target Buffers	Precise & efficient abstractions, for • LRU [Ferdinand, 1999] Not-so-precise but efficient abstractions, for • FIFO, PLRU, MRU [Grund and Reineke, 2008-2011]
Complex Pipelines	Precise but very inefficient; little abstraction Major challenge: timing anomalies
Shared resources, e.g. busses, shared caches, DRAM	No realistic approaches yet Major challenge: interference between hardware threads → execution time depends on corunning tasks

Timing Anomalies

Timing Anomaly =

Local worst-case does not imply Global worst-case

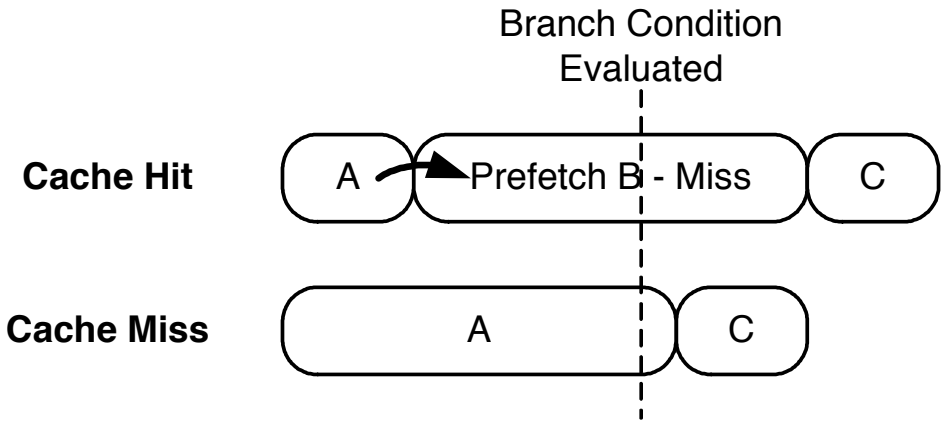


Scheduling Anomaly

Timing Anomalies

Timing Anomaly =

Local worst-case does not imply Global worst-case



Speculation Anomaly

The Design Challenge

Wanted:

Multi-/many-core architecture with

- No **timing anomalies**

→ precise & efficient analysis of individual cores

- **Temporal isolation** between cores

→ independent/incremental development & analysis

and high performance!

Approaches to the Design Challenge

At the level of individual cores:

- Simple in-order pipelines, with static or no branch prediction
- Scratchpad Memories or LRU Caches

Software-controlled caches

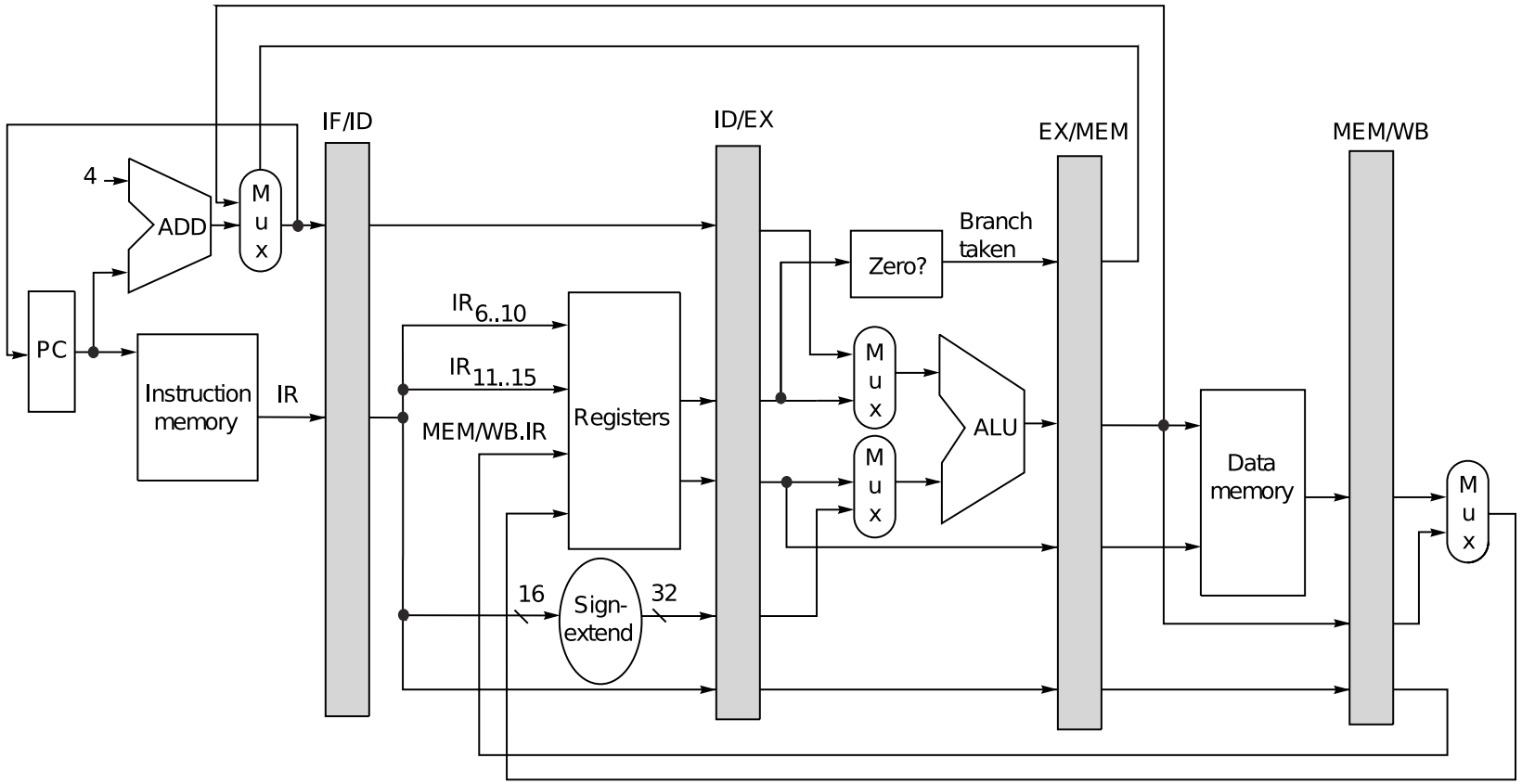
Approaches to the Design Challenge

For resources shared among multiple cores:

- Temporal partitioning, e.g.
  - TDMA arbitration of buses, shared memories
  - Thread-interleaved pipeline in PRET
- Spatial partitioning, e.g.
  - Partition shared caches
  - Partition shared DRAM

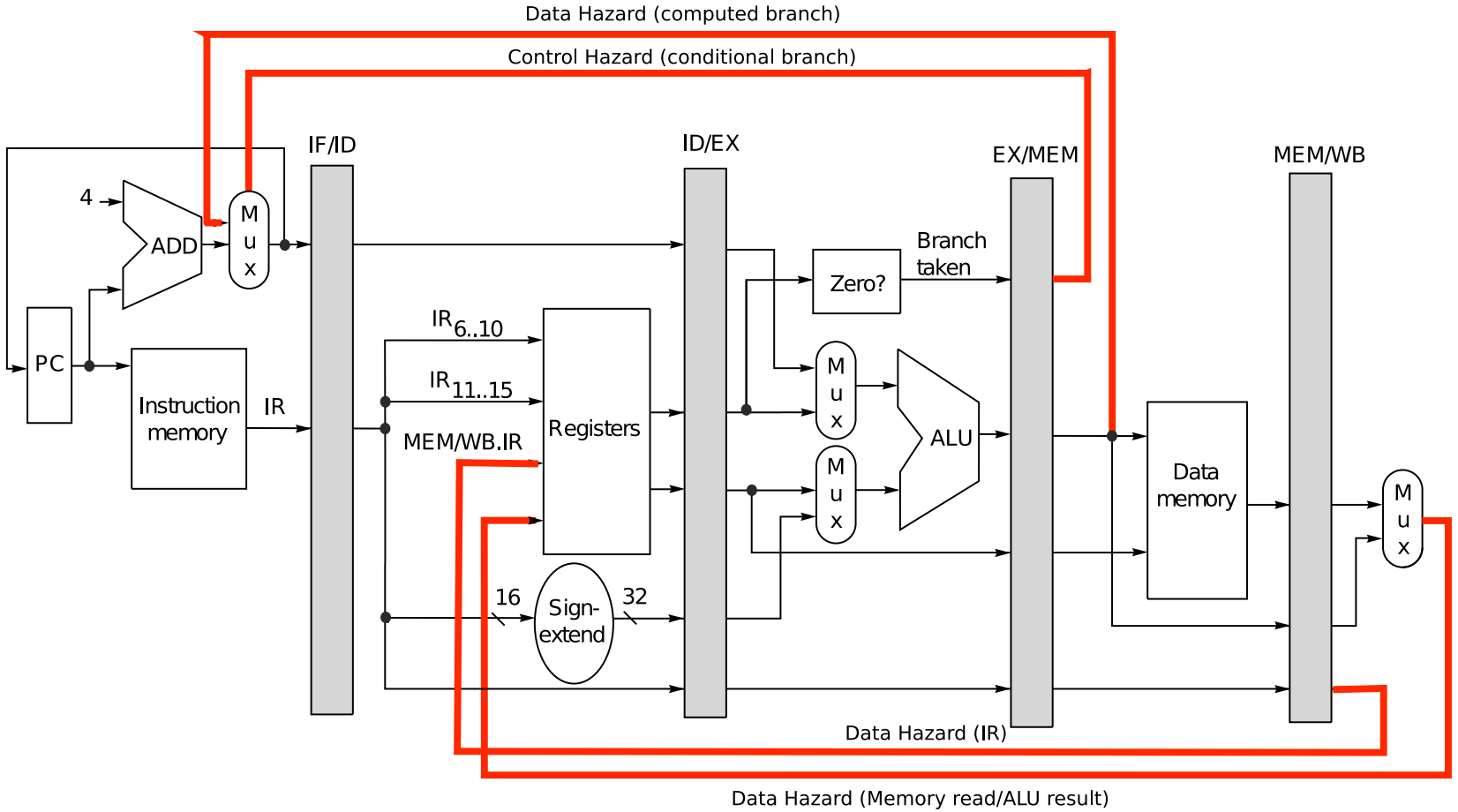
→ Temporal isolation

Design Challenge: Predictable Pipelining



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

Pipelining: Hazards



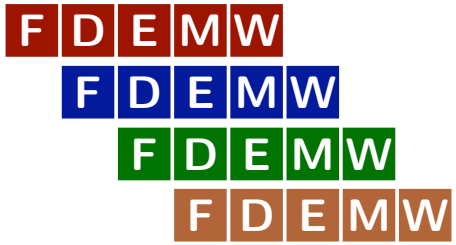
from Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 2007.

Forwarding helps, but not all the time…

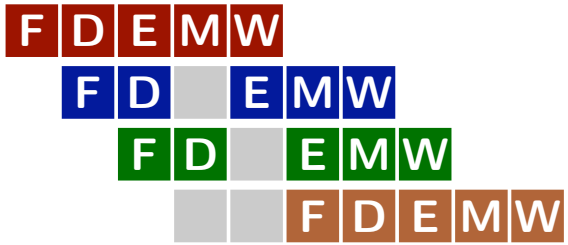
```
LD R1, 45(r2)
DADD R5, R1, R7
BE R5, R3, R0
ST R5, 48(R2)
```

Unpipelined

The Dream



The Reality



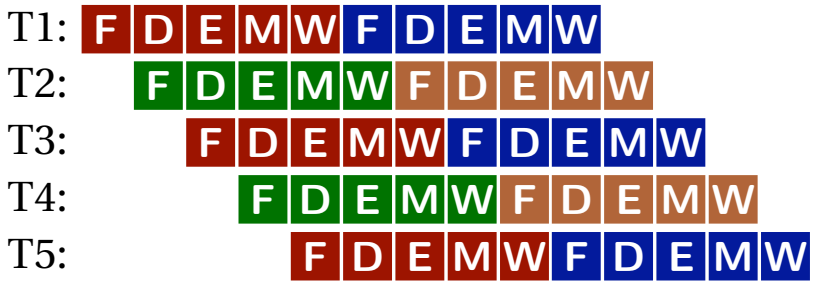
Memory Hazard

Data Hazard

Branch Hazard

Solution: PTARM Thread-interleaved Pipelines [Lickly et al., CASES 2008]

+



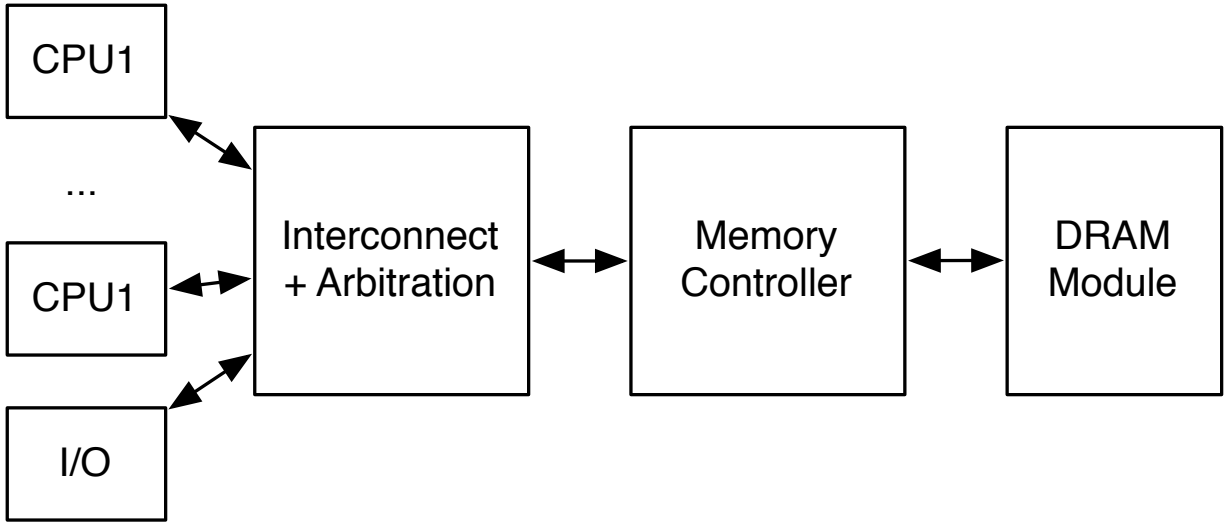
Each thread occupies only one stage of the pipeline at a time
- → No hazards; perfect utilization of pipeline
- → Simple hardware implementation (no forwarding, etc.)
- → Each instruction takes the same amount of time
- → Temporal isolation between different hardware threads

Drawback: reduced single-thread performance

Design Challenge: DRAM Controller

Translates sequences of memory accesses by Clients (CPUs and I/O) into **legal** sequences of DRAM commands

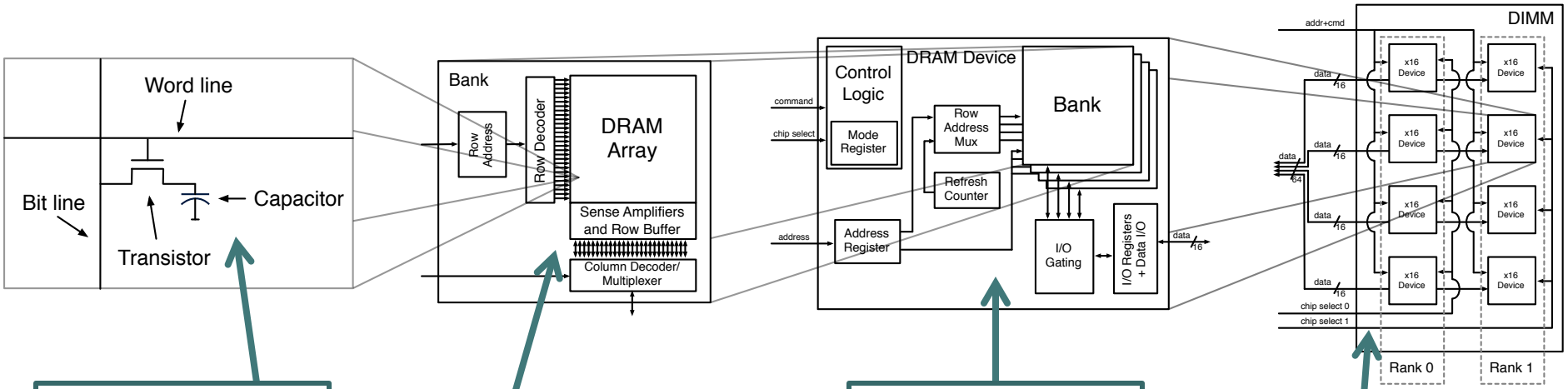
- Needs to obey all timing constraints
- Needs to insert refresh commands sufficiently often
- Needs to translate “physical” memory addresses into row/column/ bank tuples



Dynamic RAM Timing Constraints

DRAM Memory Controllers have to conform to different timing constraints that define minimal distances between consecutive DRAM commands.

Almost all of these constraints are due to the sharing of resources at different levels of the hierarchy:



Needs to insert refresh commands sufficiently often

Rows within a bank share sense amplifiers

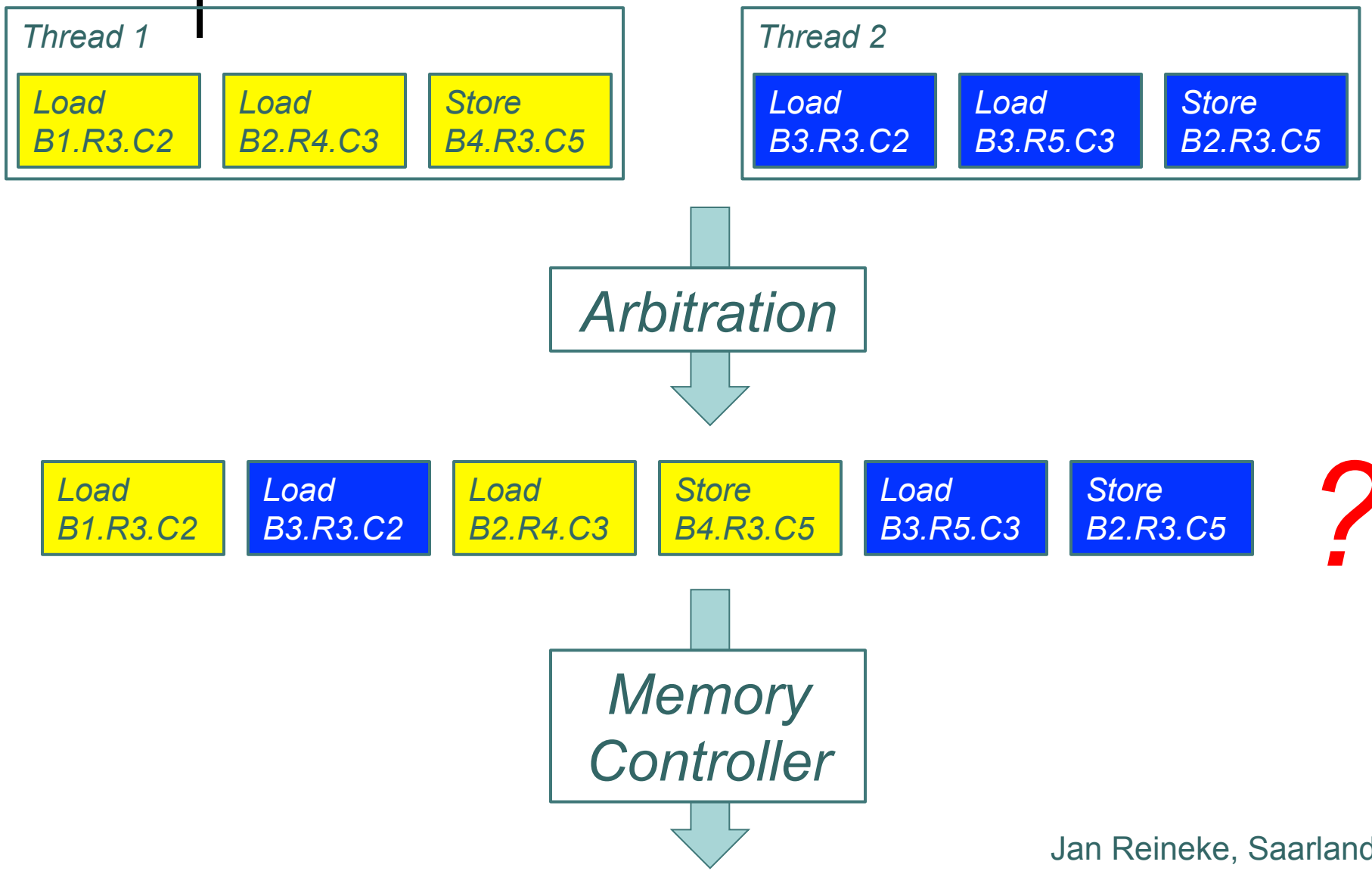
Banks within a DRAM device share I/O gating and control logic

Different ranks share data/address/ command busses

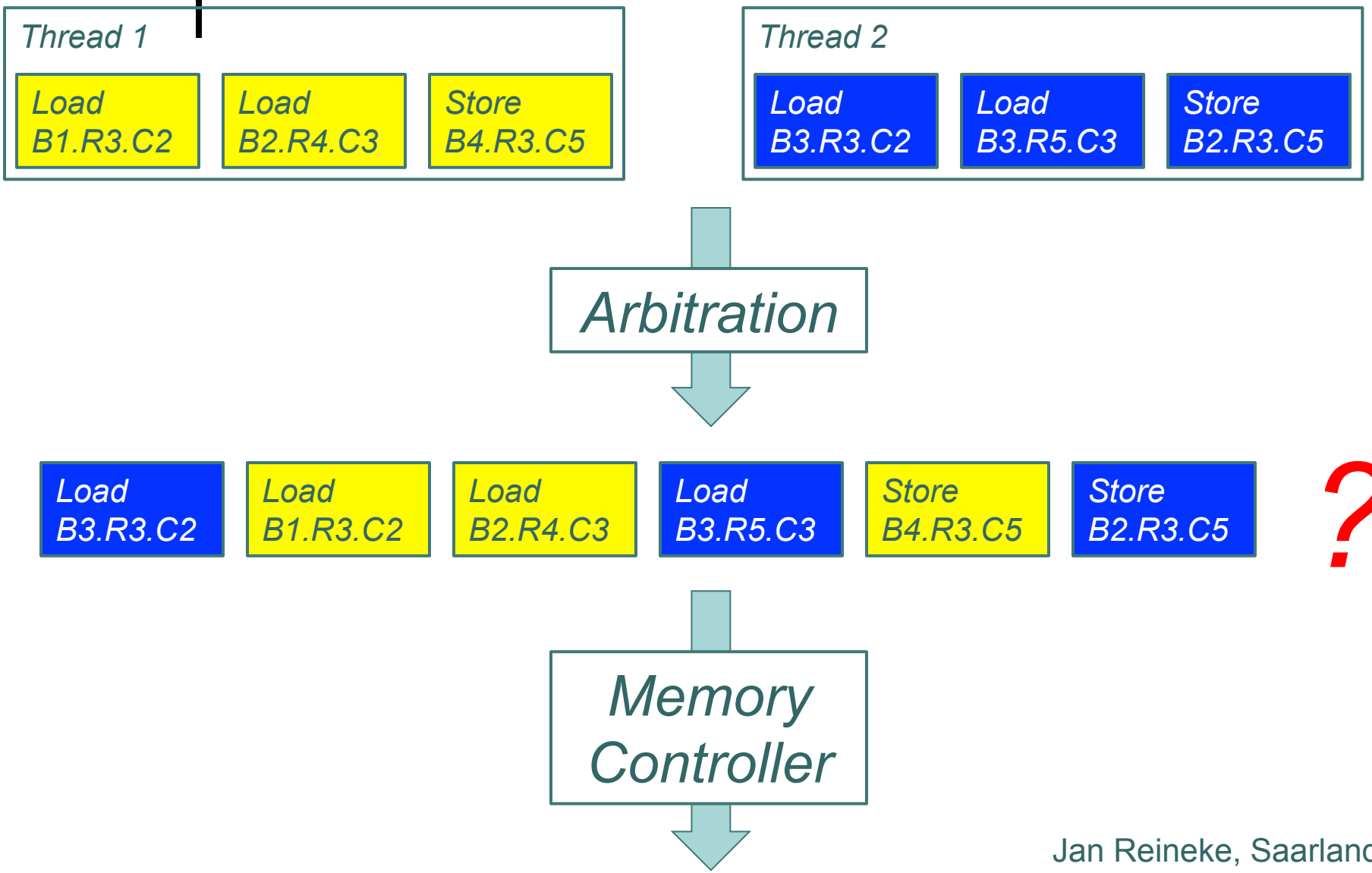
General-Purpose DRAM Controllers

- Schedule DRAM commands dynamically
- Timing hard to predict even for single client:
  - Timing of request depends on past requests:
    - Request to same/different bank?
    - Request to open/closed row within bank?
    - Controller might reorder requests to minimize latency
  - Controllers dynamically schedule refreshes
- No temporal isolation. Timing depends on behavior of other clients:
  - They influence sequence of “past requests”
  - Arbitration may or may not provide guarantees

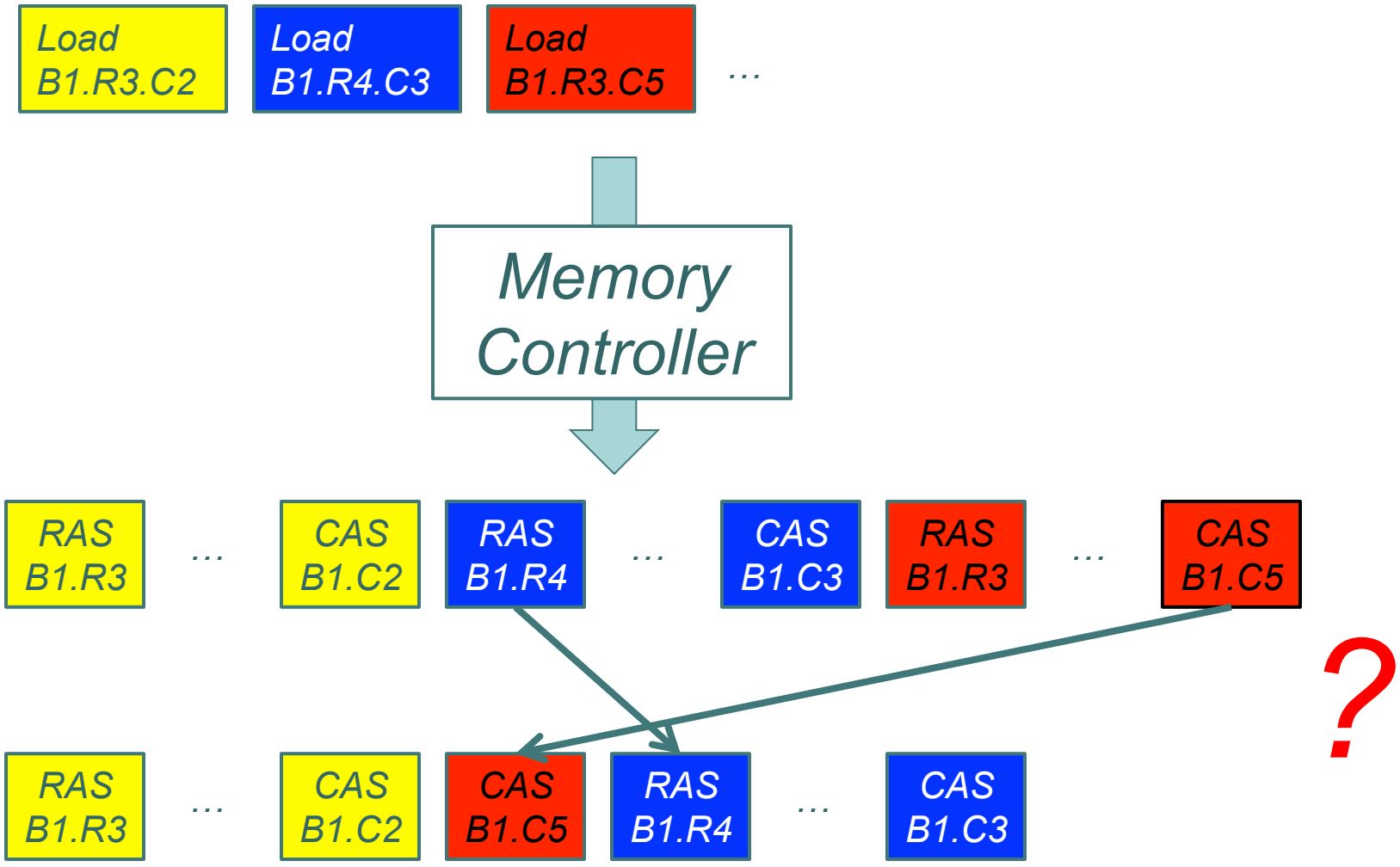
General-Purpose DRAM Controllers



General-Purpose DRAM Controllers

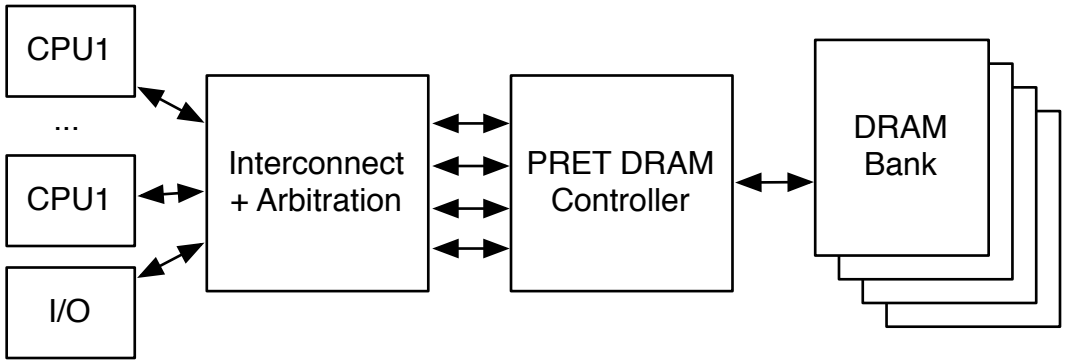


General-Purpose DRAM Controllers



PRET DRAM Controller: Three Innovations [Reineke et al., CODES+ISSS 2011]

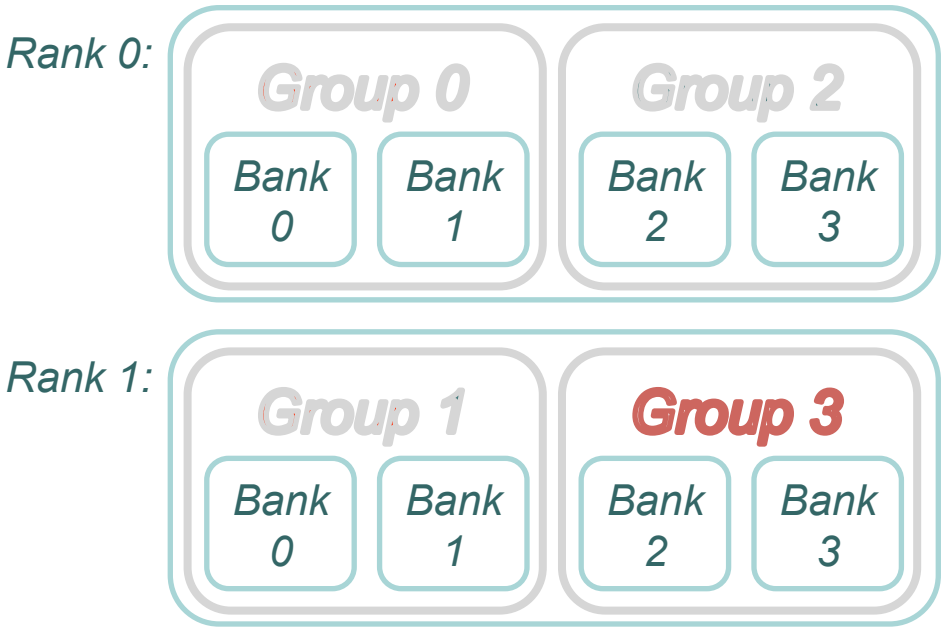
- Expose internal structure of DRAM devices:
  - Expose individual banks within DRAM device as multiple independent resources



- Defer refreshes to the end of transactions
  - Allows to hide refresh latency
- Perform refreshes “manually”:
  - Replace standard refresh command with multiple reads

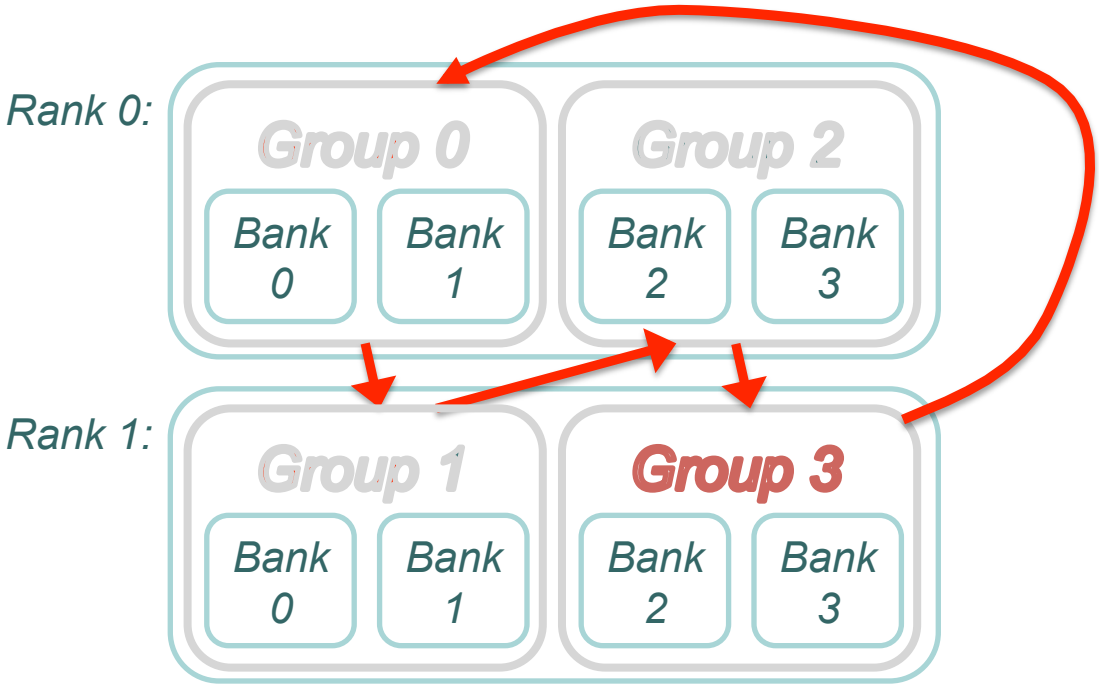
PRET DRAM Controller: Exploiting Internal Structure of DRAM Module

- Consists of 4-8 banks in 1-2 ranks
  - Share only command and data bus, otherwise independent
- Partition banks into four groups in alternating ranks
- Cycle through groups in a time-triggered fashion



PRET DRAM Controller: Exploiting Internal Structure of DRAM Module

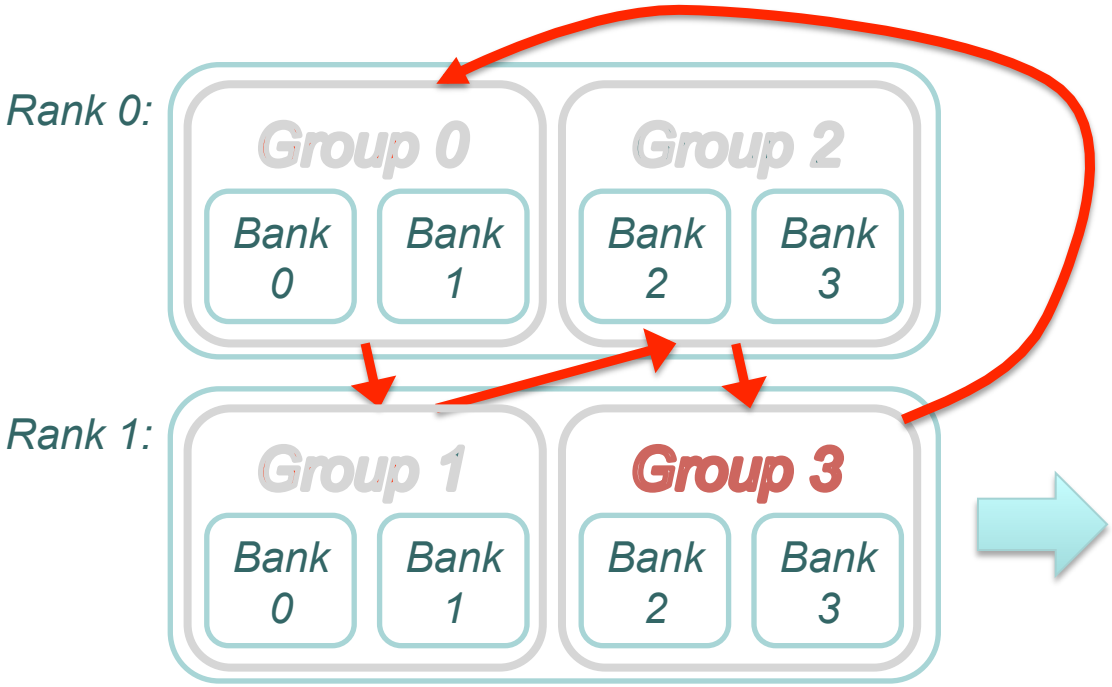
- Consists of 4-8 banks in 1-2 ranks
  - Share only command and data bus, otherwise independent
- Partition banks into four groups in alternating ranks
- Cycle through groups in a time-triggered fashion



- *Successive accesses to same group obey timing constraints*
- *Reads/writes to different groups do not interfere*

PRET DRAM Controller: Exploiting Internal Structure of DRAM Module

- Consists of 4-8 banks in 1-2 ranks
  - Share only command and data bus, otherwise independent
- Partition banks into four groups in alternating ranks
- Cycle through groups in a time-triggered fashion

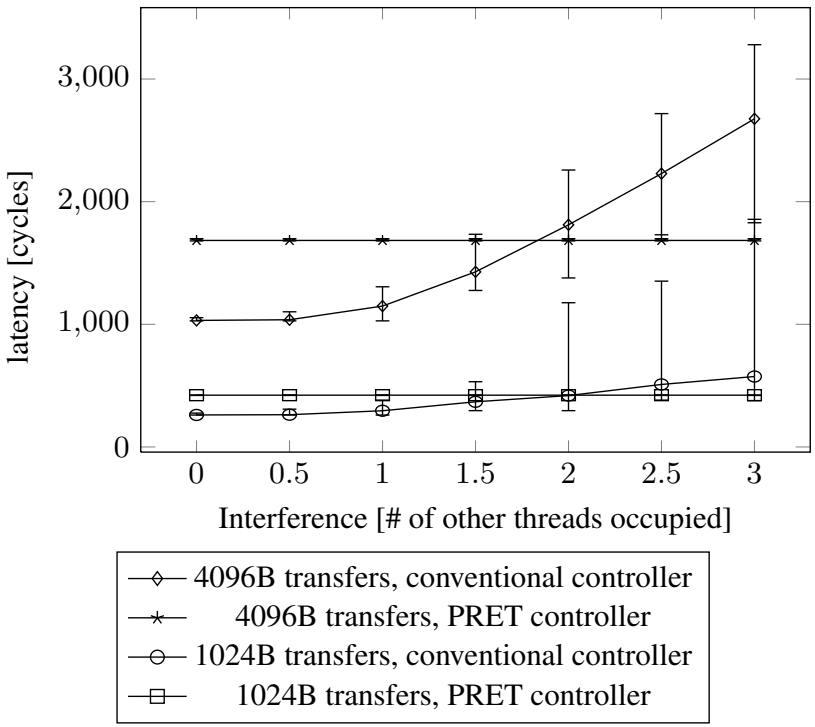


- *Successive accesses to same group obey timing constraints*
- *Reads/writes to different groups do not interfere*

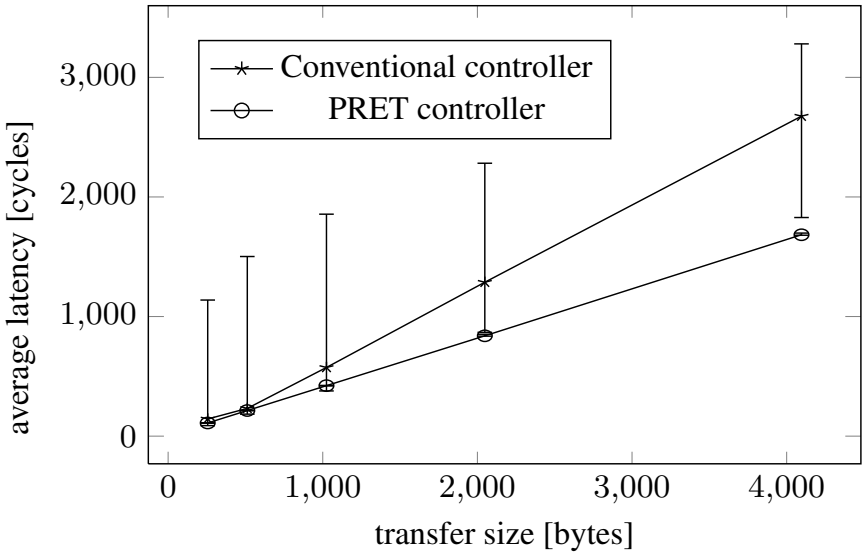
Provides four independent and predictable resources

Conventional DRAM Controller (DRAMSim2) vs PRET DRAM Controller: Latency Evaluation

Varying interference for fixed transfer size:



Varying transfer size at maximal interference:



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

Emerging Challenge: Microarchitecture Selection & Configuration

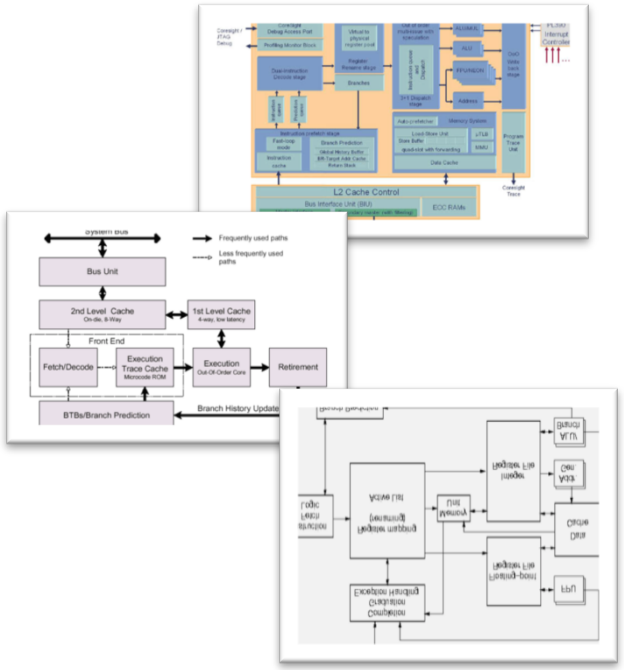
```
// Perform the convolution.
for (int i=0; i<10; i++) {
  x[i] = a[i]*b[j-i];
  // Notify listeners.
  notify(x[i]);
}
```

Embedded Software

with

Timing Requirements

?



Family of Microarchitectures = Platform

Emerging Challenge: Microarchitecture Selection & Configuration

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

Embedded Software

with

Timing Requirements

?

Choices:

- *Processor frequency*
- *Sizes and latencies of local memories*
- *Latency and bandwidth of interconnect*
- *Presence of floating-point unit*
- *…*

Family of Microarchitectures = Platform

Emerging Challenge: Microarchitecture Selection & Configuration

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

Em

Select a microarchitecture that
a) satisfies all timing requirements, and
b) minimizes cost/size/energy.

Timing Requirements

Choices:

- *Processor frequency*
- *Sizes and latencies*
- *Presence of floating-point unit*
- *…*

Family of Microarchitectures = Platform

Conclusions

Challenges in

modeling

analysis

design

remain.

Conclusions

Challenges in		*Progress based on*
modeling	→	*automation*
analysis	→	*abstraction*
design	→	*partitioning*
remain.		*has been made.*

Conclusions

Challenges in		*Progress based on*
modeling	→	*automation*
analysis	→	*abstraction*
design	→	*partitioning*
remain.		*has been made.*

Thank you for your attention!