NTNU

Det skapende universitet

TDT4255 Computer Design

Lecture 7: Memory Operations in OOO Architectures

Magnus Jahre

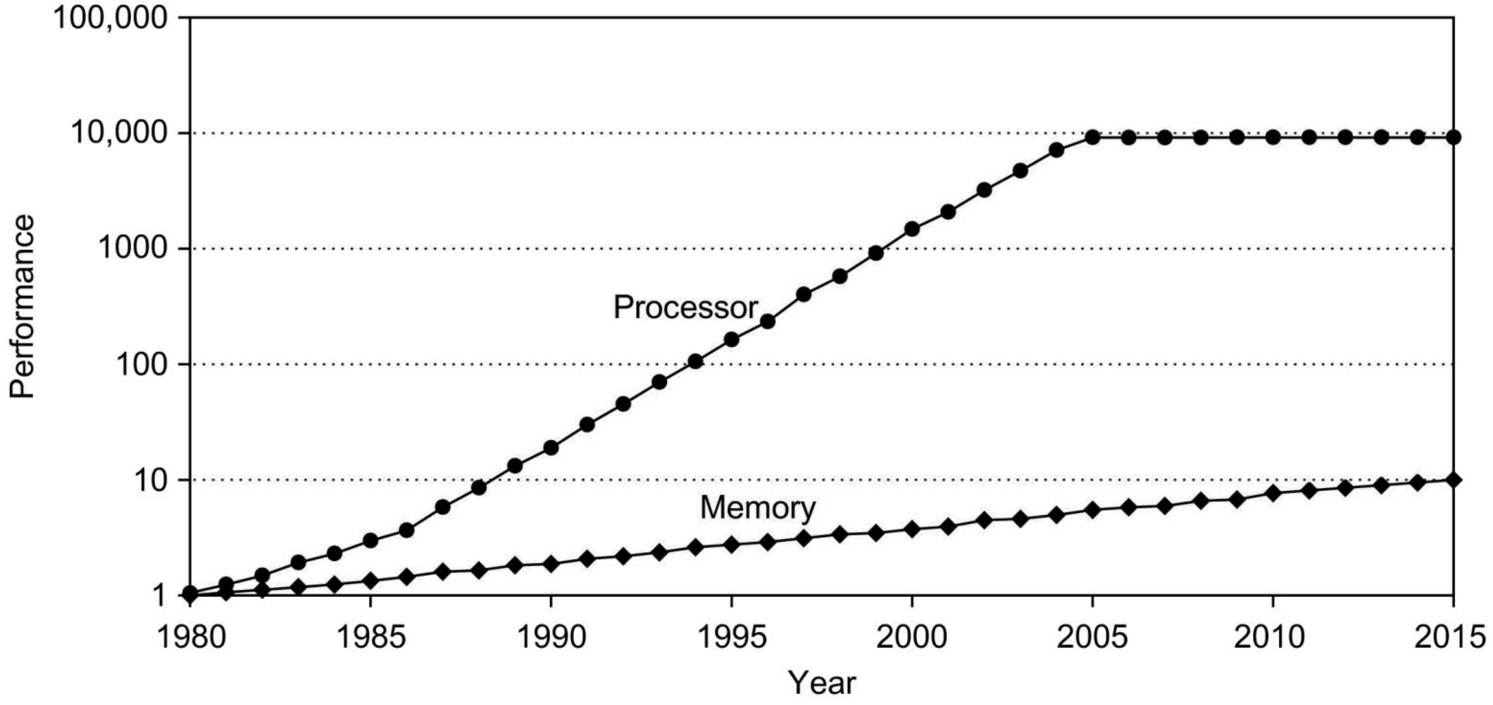
Outline

- Bridges the gap between Appendix B and Chapter 3

MEMORY HIERARCHY BASICS

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

The Memory Wall



Memory is much slower than the processor(s): We need to try to hide this latency!

Important concepts

- Locality: spatial, temporal
- Caches: direct-mapped, associative
- Replacement policy
- Write through/back
- Write allocate/non-allocate
- Virtual memory
- TLB

Goal of memory hierarchy

- Hide memory latency
- Give the illusion of infinite capacity
- At relatively low cost

Memory hierarchy

Component	Technology	Bandwidth	Latency	Cost per gigabyte (USD)
Hard drive	Magnetic	10+ MB/s	10ms	< 1
Main memory	DRAM	2+ GB/s	50+ ns	< 200
On-chip L2 cache	SRAM	10+ GB/s	2+ ns	<100.000
On-chip L1 cache	SRAM	50+ GB/s	300+ ps	<100.000
Register file	SRAM w/ multiple read/write ports	200+ GB/s	300+ ps	>10.000.000 (?)

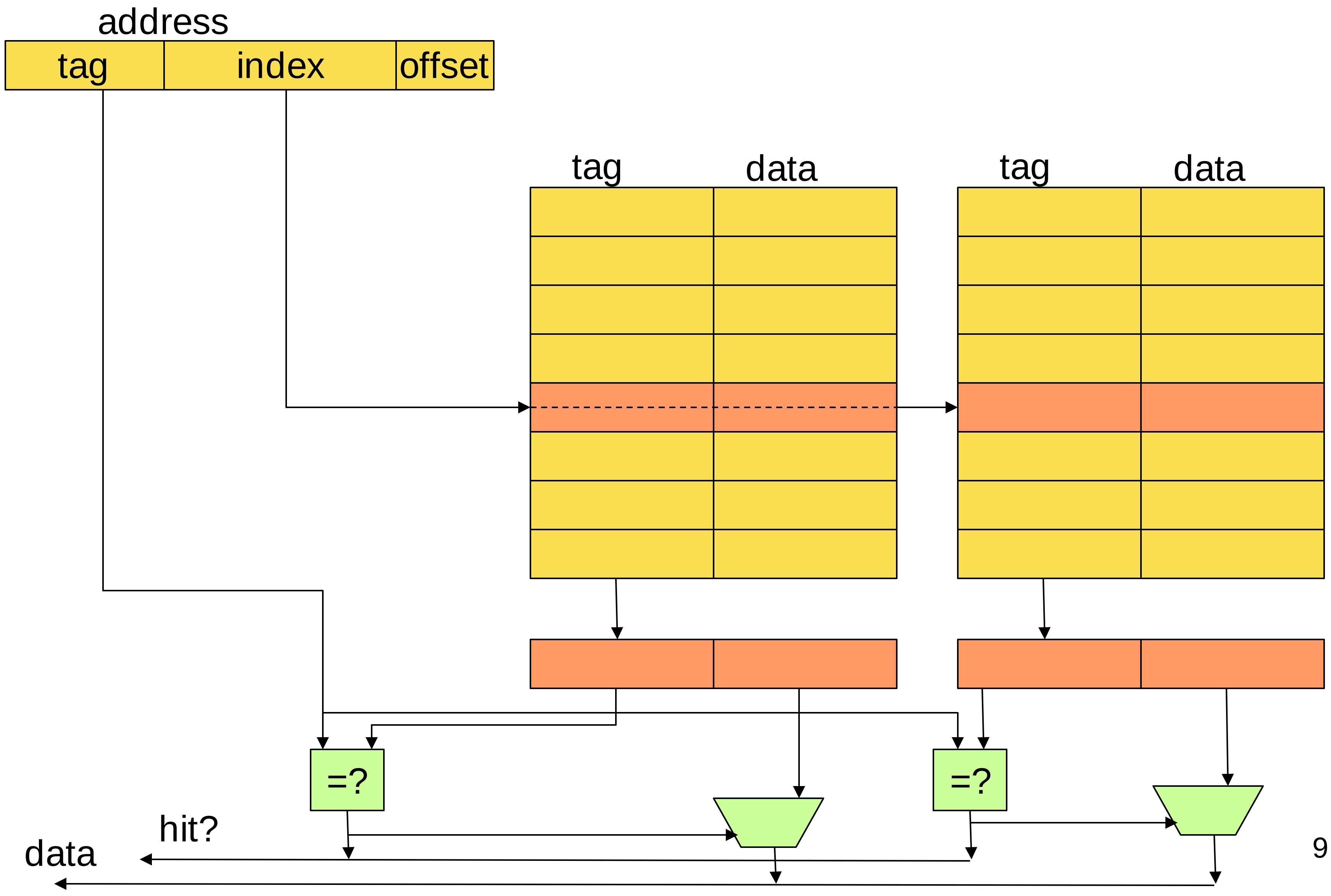
Locality

- Memory hierarchy exploits locality
- Temporal locality
  - Accesses to the same location are likely to occur in near time
- Spatial locality
  - Accesses to nearby locations are likely to occur in near time
- In both the data and instruction stream
- The case for nearly all computer programs

Exploiting locality

- Frequently used data and instructions are stored close to the processor
  - In registers
    - done by the compiler
  - If not possible, in cache hierarchy (L1, L2, etc.)
    - done by hardware
- Infrequently used data and instructions are stored far away from the processor
  - Main memory or disk

Set-associative cache



Replacement policy

- In set-associative and fully associative caches
- Well-known policies
  - First-in first-out (FIFO)
  - Least recently used (LRU)
    - Not most recently used (NMRU)
  - Random

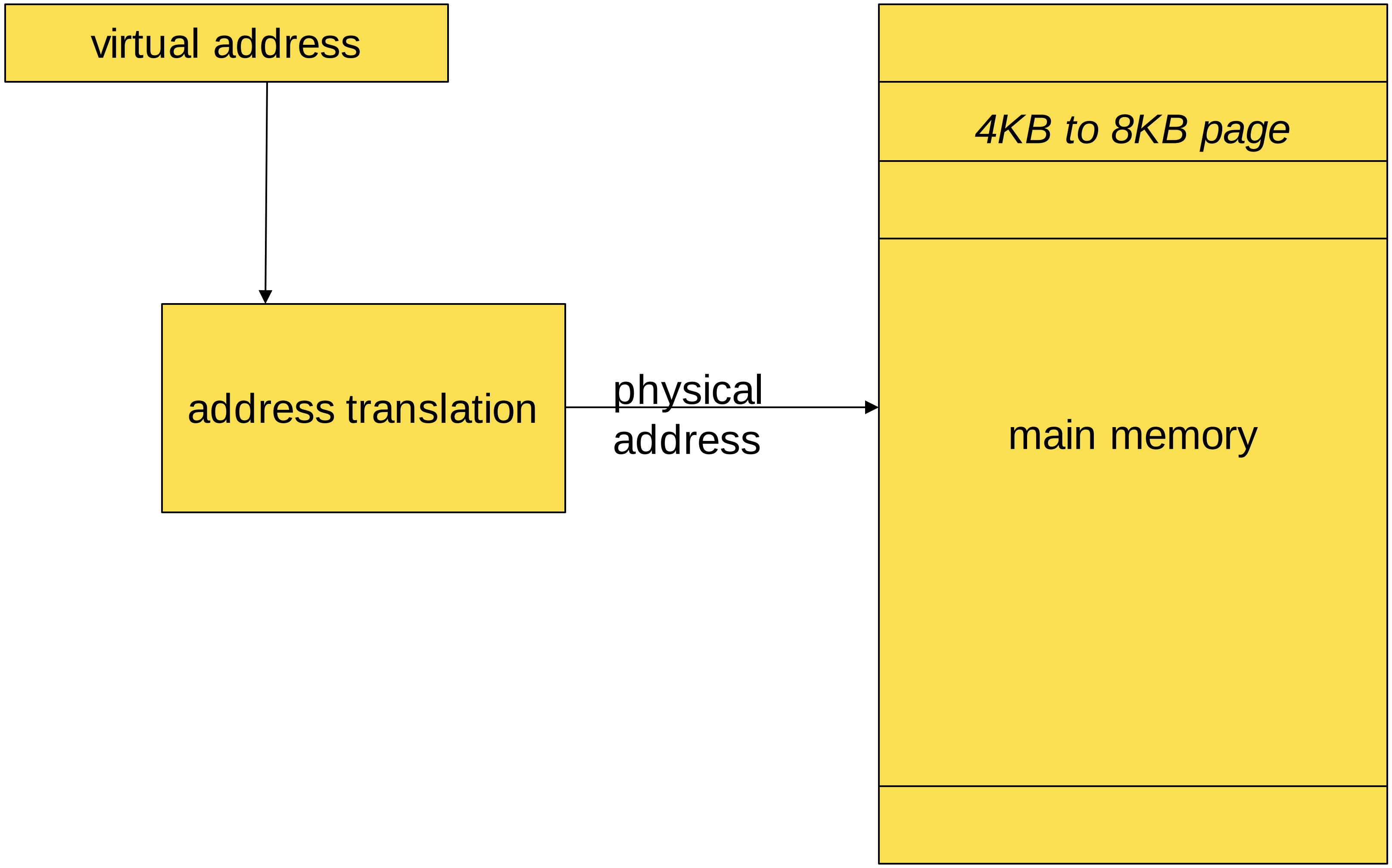
Write operations and caches

- Write-through vs. write-back
  - Write-through: when writing data to the cache, also write the data to the next level in the hierarchy (L1, L2, L3 cache or main memory)
    - Bandwidth hungry
  - Write-back: when writing data to the cache, do not write data to the next level in the hierarchy, but mark the cache block as '*dirty*'
    - Data is written back upon eviction
- Write-allocate vs. write no-allocate
  - Write-allocate: allocate data in cache
  - Write no-allocate: do not allocate in cache
- Each level of cache can be either write-through/back and write-allocate/no-allocate

Virtual memory

- Software sees a 32/64-bit address space
- Physical memory is commonly much smaller than the virtual address space
- Virtual memory translates virtual addresses to physical addresses
- Provide the illusion that the complete virtual memory is available in a much smaller physical memory
- And, multiple processes have this same view — calls for *time sharing* and *demand paging*

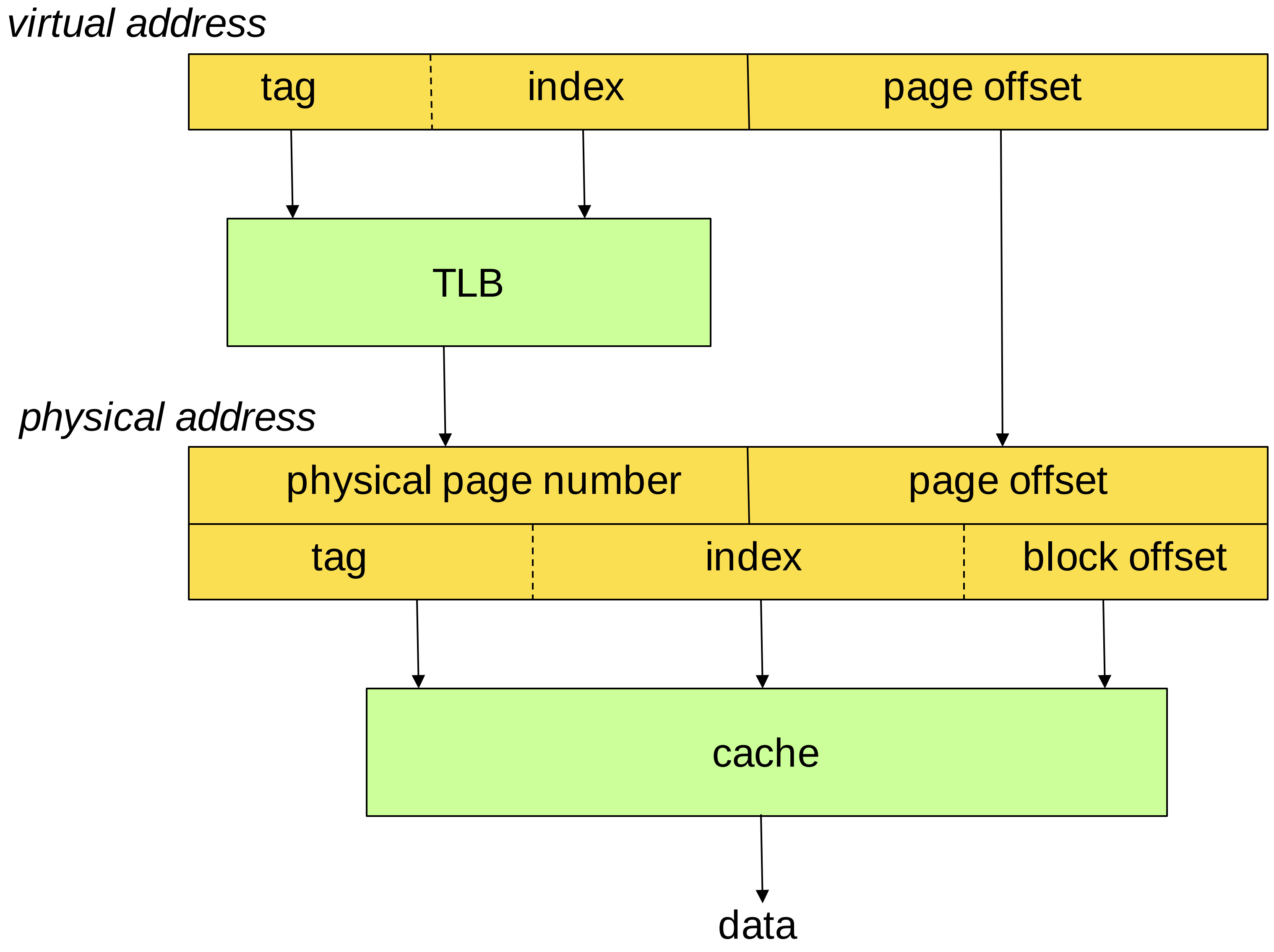
Address translation



Keeping track of page tables

- Forward page tables
- Inverted page tables
  - comes in handy wrt cache coherence (take TDT4260 in the spring semester)
- TLB
  - Translation lookaside buffer
  - Small hardware structure
    - Highly associative w/ typically 64 entries
    - Is essentially a cache for the page tables in memory
  - For both instructions (I-TLB) and data (D-TLB)
  - In case of a TLB miss
    - Read the address translation from the page table in memory
      - Can be done in HW or SW
    - Note that the page table itself may not be in main memory…

Cache and TLB



Demand paging

- Hardware figures out whether page is available in main memory
- If not, software needs to load the page from disk
  - Takes about 10 ms
  - Block the processor? No
  - Generate an exception (***page fault***); OS schedules the process out; OS identifies a page to be replaced; OS loads new page from disk; in the meanwhile, other processes run; the process is set runnable again

Memory protection

- Physical memory contains multiple pages from multiple processes
- Process A should not change memory state of process B
  - Although same virtual address, physical address is different
- In particular cases, we may want process A to change memory state of process B
  - Shared memory between processes or shared libraries
  - Different virtual addresses, same physical address
- Certain memory accesses may not be wanted a single process either: protection bits (e.g., read-only pages, etc.) — see courses on Operating Systems

MEMORY OPERATIONS IN AN OOO PROCESSOR

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

Memory operation

- Three steps
  - Address calculation
    - Typically: addition of register value with offset
    - Done in the virtual address space
  - Address translation
    - From virtual to physical
  - Memory access
    - Read (load) or write (store) memory location

Loads/stores in OoO processor

- Load needs to wait for the register with the address info
- Store needs to wait for two registers (address + value)
- Pipelined execution on FU
  - First stage: address calculation
  - Second stage: address translation
    - Access TLB (may lead to a TLB miss, etc.)
  - Third stage
    - Load: read value from cache/memory and write into a register
    - Store: write value to store queue
      - is later written into store buffer upon completion
      - is yet later written to cache/memory upon retirement

Loads/stores and speculative execution

Is not a problem (for correctness)

- Loads: read values along predicted paths, which are added into the caches; do not handle page faults unless along correct path
- Stores: memory state is not updated with speculative values because of in-order completion from store queue

Note: Changing application-visible state (e.g., caches) create the possibility for side-channel attacks (e.g., Spectre and Meltdown)

Dependences between memory operations

Dependences between memory operations

- RAW
- WAR
- WAW

Writes happen in program order (in-order completion of stores), hence WAW and WAR hazards cannot occur

RAW dependences are a potential problem

because of OOO execution: loads may read data values before older stores have written their data values

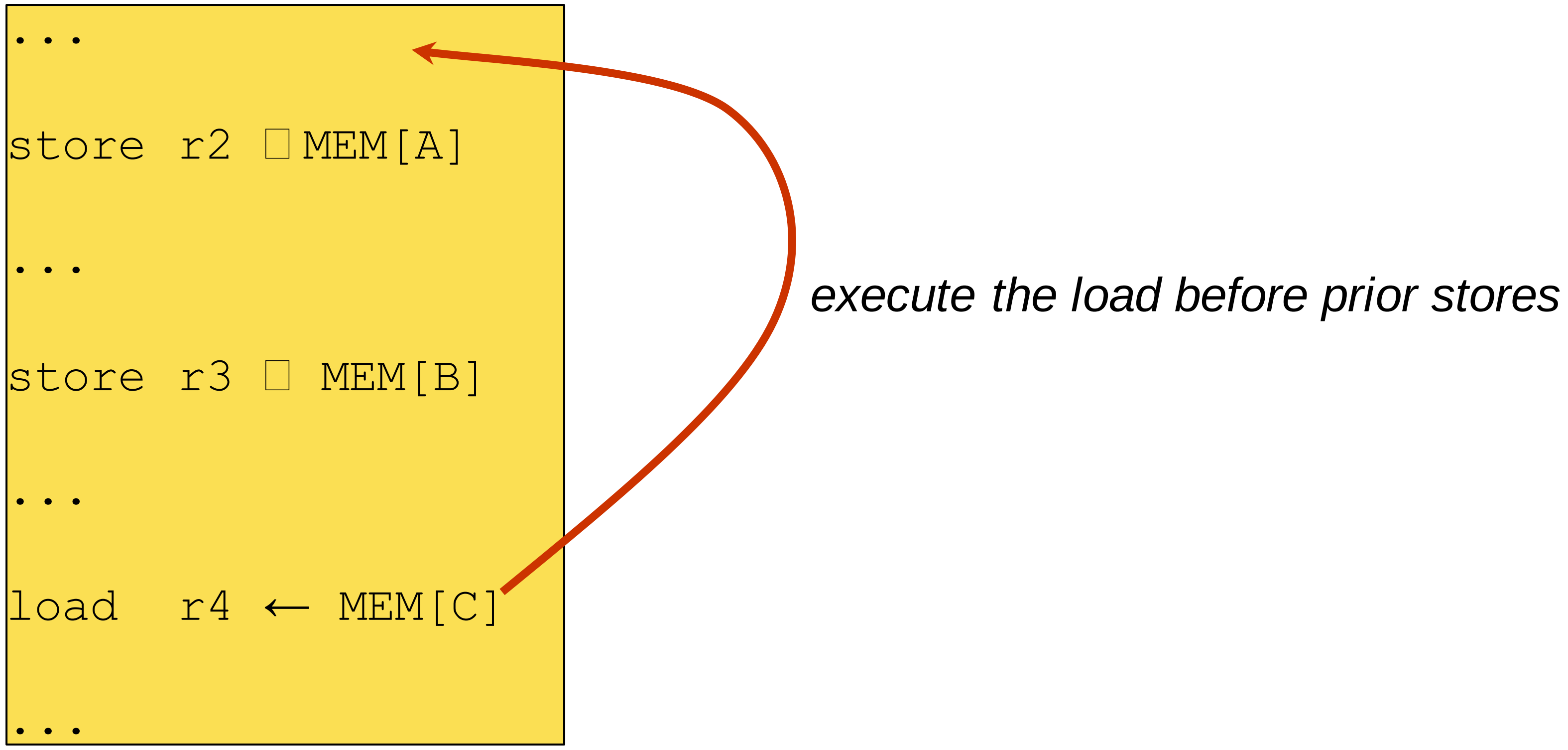
Out-of-order execution of loads?

```
...

store r2 □MEM[A]

...

store r3 □ MEM[B]

...

load  r4 ← MEM[C]

...
```

OoO execution of loads

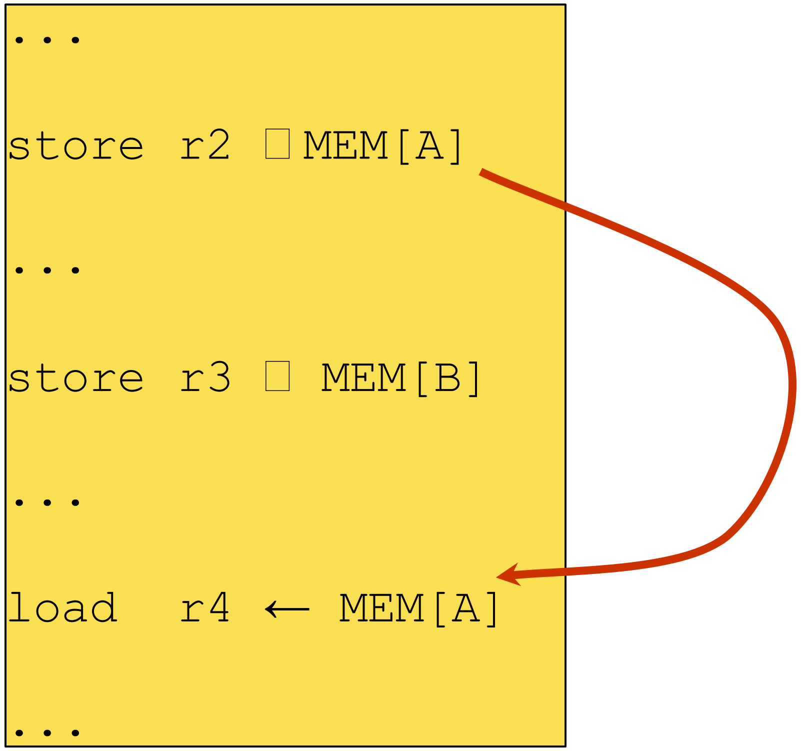
- Loads are often the beginning of a chain of dependent instructions
- Renaming (alike register renaming) is impossible because addresses are not known a priori (in the front-end pipeline)
- Two techniques to accelerate load execution:
  - Load bypassing
  - Load forwarding

Load bypassing



```
...
store r2 □ MEM[A]
...
store r3 □  MEM[B]
...
load   r4 ←  MEM[C]
...
```

Load forwarding



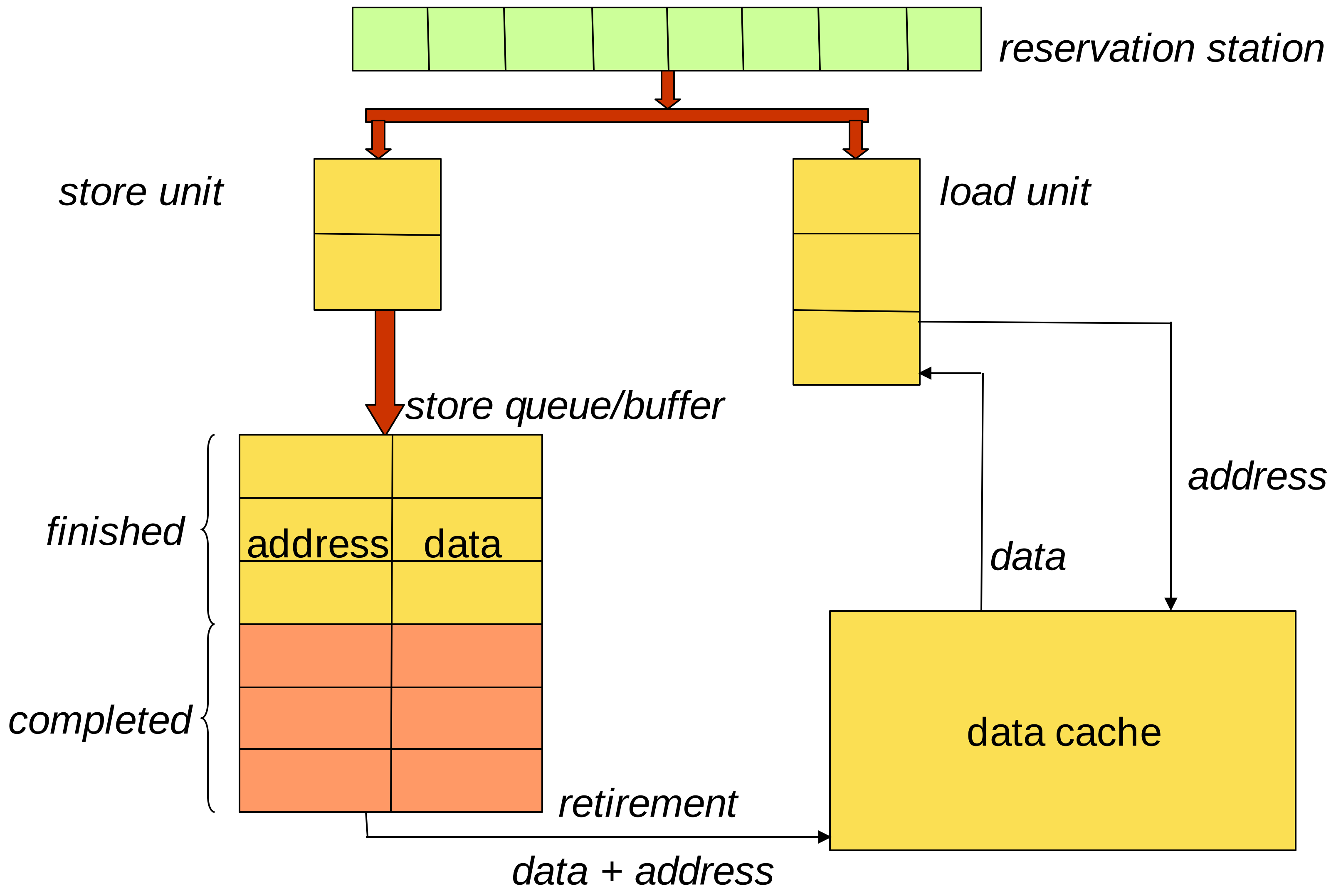
```
...

store r2 □MEM[A]

...

store r3 □ MEM[B]

...

load   r4 ← MEM[A]

...
```

RAW dependence;
forward the data from the store the load without accessing the memory hierarchy

How to implement in OoO processor?

- Non-trivial…
- Let’s proceed in two steps:
  - In-order execution of loads/stores
  - Out-of-order execution of load/stores
- Performance benefit
  - 11% to 19% through load bypassing
  - additional 1% to 4% through load forwarding

Organization



How it works

- We assume in-order execution of memory operations (*for now*)
- Loads always get priority over stores, because loads are typically on the critical path, and store latency can be hidden to some extent
- Store queue/buffer
  - Finished: both speculative and non-speculative instructions – ***"store queue"***
    - Can be nullified if along mispredicted path
  - Completed: non-speculative stores – ***"store buffer"***
    - Are fully executed from an architecture (HW/SW interface) point of view

Load bypassing & forwarding

- What's the problem?
  - Most recent value may not be in caches/memory but in the store queue/buffer
- Hence, we need to search the store queue/buffer for the most recent value of that same memory location
- If a hit in store queue/buffer, forward the data to the load → **Load forwarding**

Store queue/buffer complexity

- Very complex hardware
  - Content addressable memory (CAM)
    - need comparators to search store queue/buffer for a particular memory address
    - possible multiple stores with the same address; need to pick the most recent one

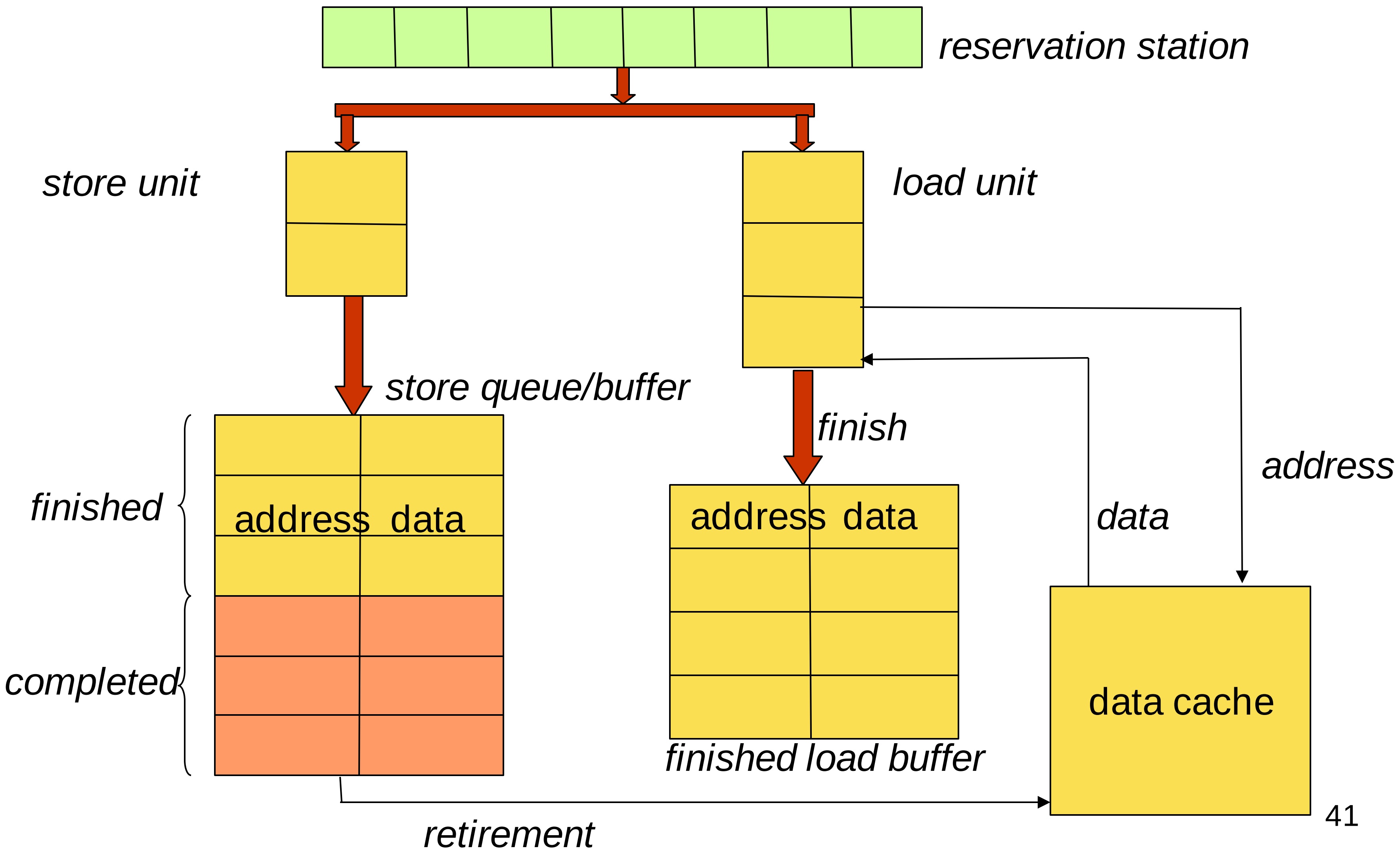
OoO execution of loads?

- In-order execution of loads guarantees that all prior stores are executed
  - Hence: we only need to search the store queue/buffer
- In-order execution limits ILP though
  - Load has to wait for all prior stores, even if a load does not depend on prior stores (which is the most likely case!)

Potential problem

- Store prior to a dependent load may not be executed when the load executes
  - Store still resides in the reservation station, or is in execution
  - Hence, value is not yet in store queue/buffer
  - Or even worse, the store address may not even be known!

OoO execution of loads



How it works

- Loads execute out of program order
  - When their input operands are available
  - At finish, address and data is stored in *finished load buffer*
  - Load entry is removed from finished load buffer at completion
- At completion of store, do an associative search in finished load buffer
  - If no match: no RAW dependence — it was okay to execute the load prior to the store
  - If match: RAW hazard — load needs to be nullified and re-executed along with all instructions after the load
    - fairly high cost — solution: *memory dependence prediction*

SUMMARY

Slides in this section are by Lieven Eeckhout, Ghent University. Reused with permission.

Summary

- Memory is much slower than the processor
  - Latency-hiding techniques are necessary to achieve high performance
  - Key techniques: Caching and executing multiple memory requests in parallel
- Loads are often performance-critical since they tend to unlock a collection of dependent instructions
  - Try to issue loads in parallel and out-of-order with respect to stores
  - Key techniques: Load bypassing and load forwarding