Design of Parallel and High-Performance Computing

Fall 2019

Lecture: Cache Coherency

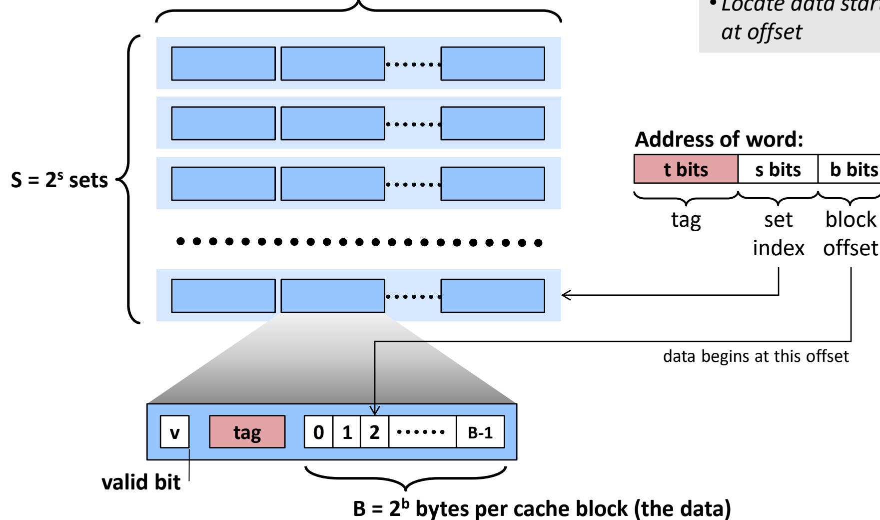
Instructor: Tal Ben-Nun & Markus Püschel

TA: Timo Schneider

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

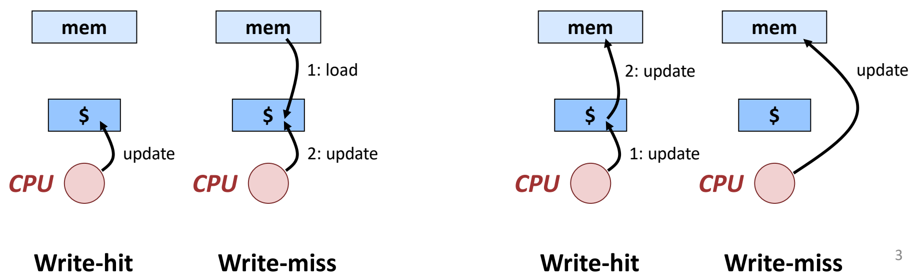


What about writes?

Put on board

- **What to do on a write-hit?**
  - ***Write-through:*** write immediately to memory
  - ***Write-back:*** defer write to memory until replacement of line
- **What to do on a write-miss?**
  - ***Write-allocate:*** load into cache, update line in cache
  - ***No-write-allocate:*** writes immediately to memory

Write-back/write-allocate (Most common) *Write-through/no-write-allocate*



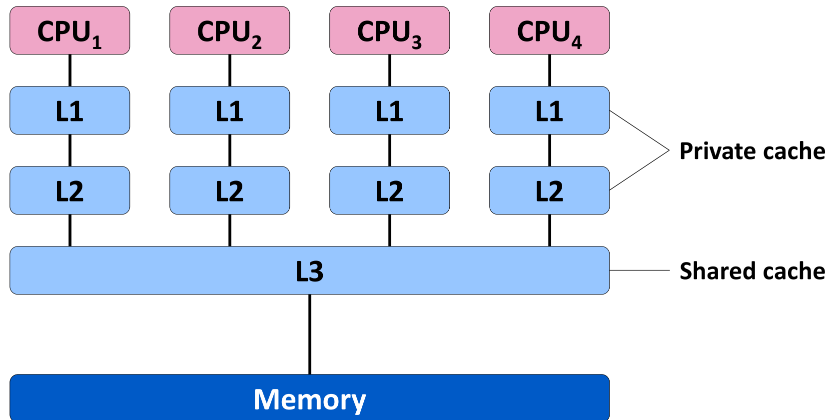
Write-hit **Write-miss** **Write-hit** **Write-miss**

Cache Architectures

- **Multi level caches** (L1, L2, …, common today)
- **Multi-port vs. single port**
- **Shared vs. private** (in multicore computers, see next slide)
- **Inclusive vs. exclusive** (usually inclusive = content of smaller caches is in larger caches in the hierarchy)
- **Write back vs. write through** (usually write-back)
- **Victim cache to reduce conflict misses**
- **…**

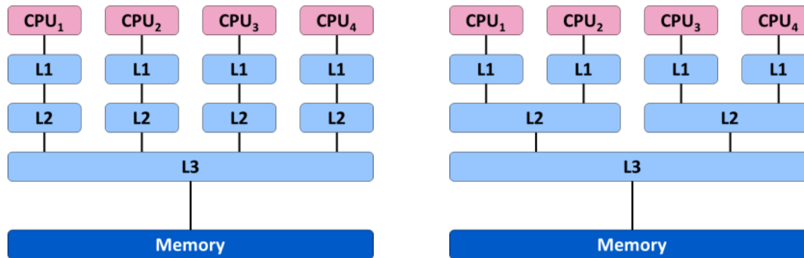
© Markus Püschel
Computer Science
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Caches in a Multicore Computer (Example)



5

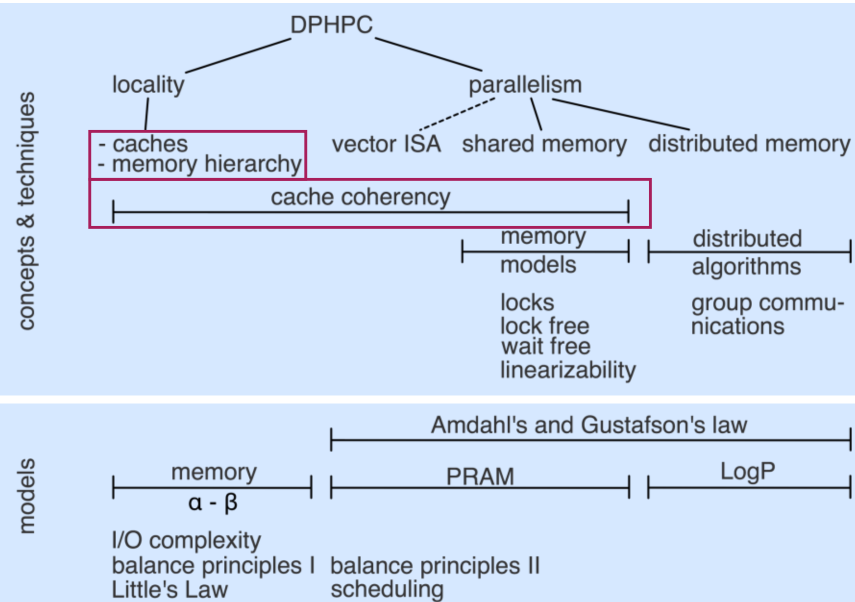
Many Variants Possible



Problem? ***Coherence!***

6

DPHPC Overview

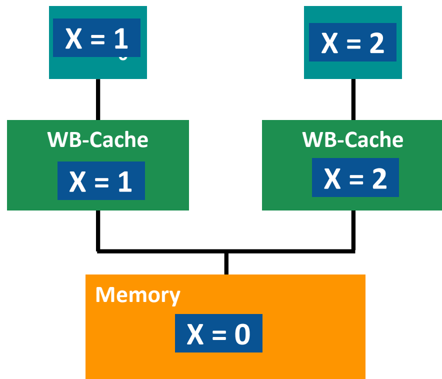


Today: Cache Coherence

- Motivation
- Terminology, snooped-based vs. directory-based
- Snooped based: MESI protocol
- MOESI
- Directory-based coherence

© Markus Püschel
Computer Science
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Write-Back Cache



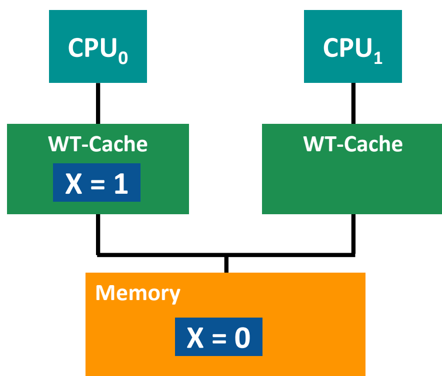
1. **CPU_0 reads X from memory**
   - loads X=0 into its cache
2. **CPU_1 reads X from memory**
   - loads X=0 into its cache
3. **CPU_0 writes X=1**
   - stores X=1 in its cache
4. **CPU_1 writes X =2**
   - stores X=2 in its cache
5. **CPU_1 writes back cache line**
   - stores X=2 in in memory
6. **CPU_0 writes back cache line**
   - stores X=1 in memory

   Later (!) store X=2 from CPU_1 lost

Requires write serialization!

9

Write-Through Cache



1. **CPU_0 reads X from memory**
   - loads X=0 into its cache
2. **CPU_1 reads X from memory**
   - loads X=0 into its cache
3. **CPU_0 writes X=1**
   - stores X=1 in its cache
   - stores X=1 in memory
4. **CPU_1 reads X from its cache**
   - loads X=0 from its cache

   Incoherent value for X on CPU_1

Requires write propagation!

10

Another Example

- **The issue in the previous examples was shared access including writes**
- **Assume C99:**

```
struct twoint {
   int a;
   int b;
}
```

- **Two threads:**
  - Thread 0: writes to a
  - Thread 1: writes to b
  - The timeline is not specified
- **Assume non-coherent write-back cache**
  - What may end up in main memory?
- ***False sharing:* two threads access different data in the same cache line**

Cache Coherence

- ***Basic problem:* threads on different cores with private caches access the same data, including with writes**
- **Cache coherence requirements**

  A memory system is coherent if it guarantees the following:
  - *Write propagation:* updates are eventually visible to all readers
  - *Write serialization:* writes to the same location must be observed in order

    Everything else: memory model issues (later)

© Markus Püschel
Computer Science
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

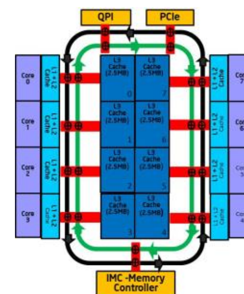
Cache Coherence Protocol

- **Programmer cannot deal with unpredictable behavior!**
- **Hardware mechanism to ensure coherence:**
  Cache controller executes protocol to maintain coherence

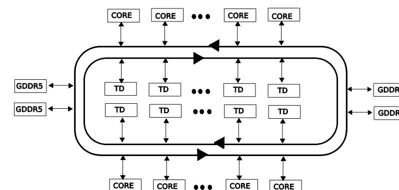
Fundamental Mechanisms

- **Snooping**
  - Shared bus or (broadcast) network
- **Directory-based**
  - Record (in a memory) information necessary to maintain coherence
    E.g., owner and state of a line etc.

Cache Coherence Approaches



- **Snooping**
  - Shared bus or (broadcast) network
  - Cache controller snoops all transactions
  - Monitors and changes state data in cache
  - Works at small scale, challenging at large-scale
  - Example: Intel Xeon etc.
- **Directory-based**
  - Record information necessary to maintain coherence
    E.g., owner and state of cache lines
  - Central or distributed directory
  - Scalable but more complex/expensive
  - Example: Intel Xeon Phi



Source: Intel

© Markus Püschel
Computer Science

Cache Coherence Parameters

- **Concerns/Goals**
  - Performance
  - Implementation cost (chip space)
  - Correctness
  - (Memory model side effects)

- **Issues**
  - Detection (when does a controller need to act)
  - Enforcement (how does a controller guarantee coherence)
  - Granularity of block sharing (typically cache block size)

Cache Coherence

- ***Basic problem:*** **threads on different cores with private caches access the same data, including with writes**
- **Cache coherence requirements**

  A memory system is coherent if it guarantees the following:
  - *Write propagation:* updates are eventually visible to all readers
  - *Write serialization:* writes to the same location must be observed in order

    Everything else: memory model issues (later)

© Markus Püschel
Computer Science

An Engineering Approach: Empirical start

- **Problem 1: stale reads**
  - Cache 1 holds value that was already modified in cache 2
  - **Solution:**

    Disallow this state

    Invalidate all remote copies before allowing a write to complete

- **Problem 2: lost update**
  - Incorrect write back of modified line writes main memory in different order from the order of the write operations or overwrites neighboring data
  - **Solution:**

    Disallow more than one modified copy

Invalidation vs. update

- **Invalidation-based:**
  - Write to a shared line has to invalidate copies in other caches
  - Subsequent writes by the same thread to the same cache line are efficient
- **Update-based:**
  - Local write updates copies in other caches
  - All sharers continue to hit cache line after one core writes
  - Supports producer-consumer pattern well
  - Many writes cause many updates

- **Hybrid forms are possible!**
- **MESI (next): Invalidation-based**

MESI Cache Coherence

- **Most common hardware implementation of coherence**
  aka. "Illinois protocol"

Each cache line has one of the following states (added as bits to line):

- **Modified (M)**
  - Local copy has been modified, no copies in other caches
  - Memory is stale
- **Exclusive (E)**
  - No copies in other caches
  - Memory is up to date
- **Shared (S)**
  - Unmodified copies *may* exist in other caches
  - Memory is up to date
- **Invalid (I)**
  - Line is not in cache

Put on board

- **Draw 4 cores with private caches, memory, bus**

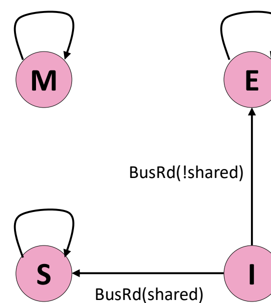
© Markus Püschel
Computer Science
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Terminology

- **Clean line:**
  - Content of cache line and main memory is identical (memory is up to date)
  - Can be evicted without write-back
- **Dirty line:**
  - Content of cache line and main memory differ (memory is stale)
  - Needs to be written back eventually
    Time depends on protocol details
- **Bus transaction:**
  - A signal on the bus that can be observed by all caches
  - Usually blocking (only one signal at a time)
- **Local (private) read/write:**
  - A load/store operation originating at a core connected to the cache

Transitions in Response to Local Reads

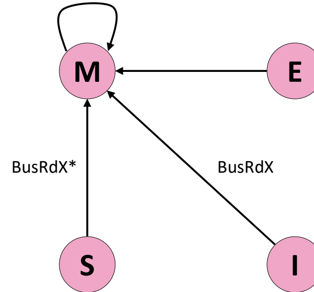
- **State is M**
  - No bus transaction, same state
- **State is E**
  - No bus transaction, same state
- **State is S**
  - No bus transaction, same state
- **State is I**
  - Generate bus read (BusRd)
    May force other cache operations (see later)
  - Other cache(s) signal "shared" if they hold a copy
  - If "shared" was signaled, go to state S
  - Otherwise, go to state E



© Markus Püschel
Computer Science
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

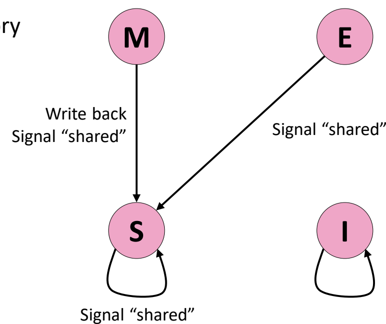
Transitions in Response to Local Writes

- **State is M**
  - No bus transaction
- **State is E**
  - No bus transaction
  - Go to state M
- **State is S**
  - Line already local & clean
  - There may be other copies
  - Generate bus read for upgrade to exclusive ownership (BusRdX*)
    May force other cache operations (see later)
  - Go to state M
- **State is I**
  - Generate bus read for exclusive ownership (BusRdX)
    May force other cache operations (see later)
  - Go to state M



Transitions in Response to Snooped BusRd

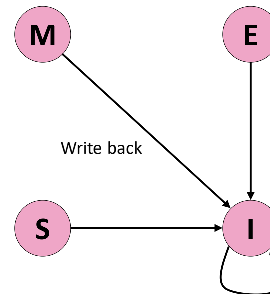
- **State is M**
  - Write cache line back to main memory
  - Signal "shared"
  - Go to state S
- **State is E**
  - Signal "shared"
  - Go to state S
- **State is S**
  - Signal "shared"
- **State is I**
  - Ignore



© Markus Püschel
Computer Science
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

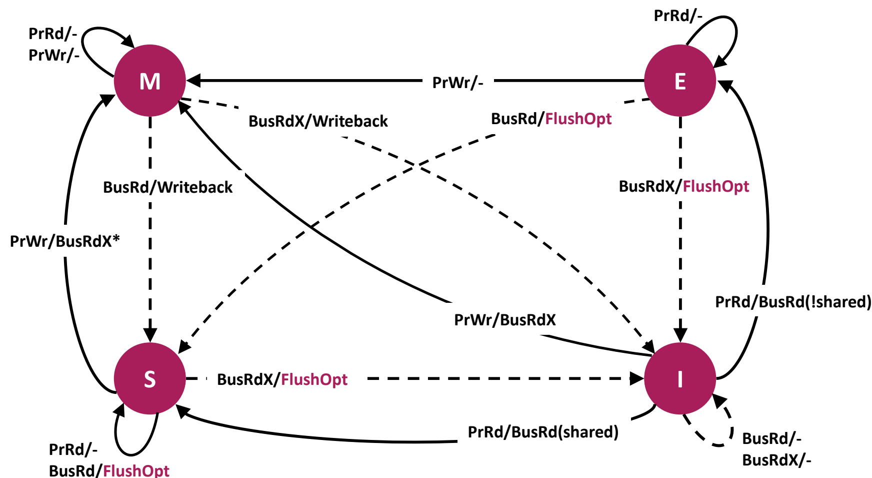
Transitions in Response to Snooped BusRdX

- **State is M**
  - Write cache line back to memory
  - Go to I (discard line)
- **State is E**
  - Go to I (discard line)
- **State is S**
  - Go to I (discard line)
- **State is I**
  - Ignore



- **BusRdX* is handled like BusRdX!**

MESI State Diagram (FSM)



Additional detail: ***FlushOpt*** = processor (may) send its copy of cache line on bus for possible faster read by other processor. On writebacks the cacheline is always on the bus for possible reading.

© Markus Püschel
Computer Science
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Small Exercise

- **Initially: all in I state**

Action	P1 state	P2 state	P3 state	Bus action	Data from
P1 reads x					
P2 reads x					
P1 writes x					
P1 reads x					
P3 writes x					

Small Exercise

- **Initially: all in I state**

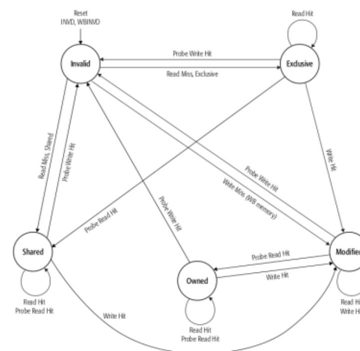
Action	P1 state	P2 state	P3 state	Bus action	Data from
P1 reads x	E	I	I	BusRd	Memory
P2 reads x	S	S	I	BusRd	Memory or Cache of P1 (FlushOpt)
P1 writes x	M	I	I	BusRdX*	Cache
P1 reads x	M	I	I	-	Cache
P3 writes x	I	I	M	BusRdX	Memory or Cache of P1 (FlushOpt)

Related Protocols: MOESI (AMD)

- **Extended MESI protocol (What could be improved?)**
- ***Cache-to-cache transfer*** **of modified cache lines**
  - Cache in M or (new) O state always transfers cache line to requesting cache
  - No need to contact (slow) main memory
- **Avoids write back when another process accesses cache line**
  - Good when cache-to-cache performance is higher than cache-to-memory
    E.g., shared last level cache!

Related Protocols: MOESI (AMD)

- **Modified (M): Modified Exclusive**
  - No copies in other caches, local copy dirty
  - Memory is stale, *cache supplies copy* (reply to BusRd*)
- **Owner (O): Modified Shared**
  - Exclusive right to make changes
  - Other S copies may exist ("dirty sharing")
  - Memory is stale, *cache supplies copy* (reply to BusRd*)
- **Exclusive (E):**
  - Same as MESI (one local copy, up to date memory)
- **Shared (S):**
  - Unmodified copy may exist in other caches
  - Memory is up to date unless an O copy exists in another cache
- **Invalid (I):**
  - Same as MESI



© Markus Püschel
Computer Science
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Multi-level caches

- **Most systems have multi-level caches (here assume 2)**
  - Problem: only "last level cache" is connected to bus or network
  - Yet, snoop requests are relevant for inner-levels of cache (L1)
  - Modifications of L1 data may not be visible at L2 (and thus the bus)
- **L1/L2 modifications**
  - On BusRd check if line is in M state in L1

    It may be in E or S in L2!
  - On BusRdX(*) send invalidations to L1
  - Everything else can be handled in L2

Directory-based cache coherence

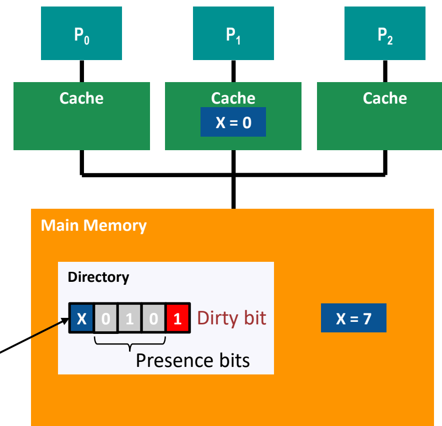
- **Snooping does not scale**
  - Bus transactions must be *globally* visible
  - Implies broadcast
- **Typical solution: tree-based (hierarchical) snooping**
  - Root becomes a bottleneck
- **Directory-based schemes are more scalable**
  - Directory (one entry for each cache line) keeps track of all owning caches
  - Point-to-point update to involved processors

    No broadcast

    Can use specialized (high-bandwidth) network, e.g., HT, QPI …

Basic Scheme (Sketch)

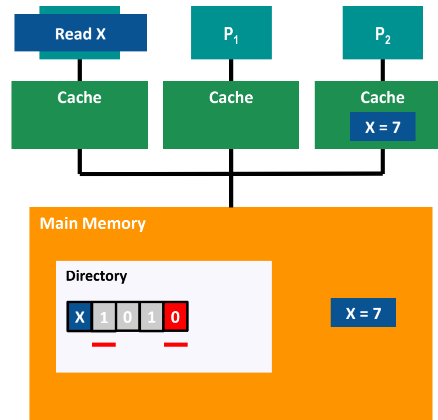
- **System with N processors P_i**
- **For each memory block (size: cache block) maintain a directory entry**
  - N presence bits
    i-th bit set = block is in cache of P_i
  - 1 dirty bit (red)
- *First proposed by Censier and Feautrier (1978)*



Cache block containing x (and adjacent data)!

Directory-based CC: Read miss

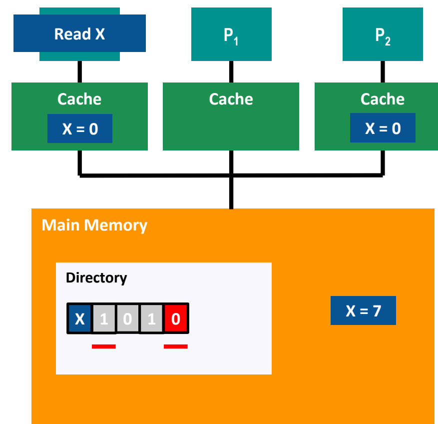
- **P_0 intends to read, misses**
- **If dirty bit (in directory) is off**
  - Read from main memory
  - Set presence[i]
  - Supply data to reader



© Markus Püschel Computer Science ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

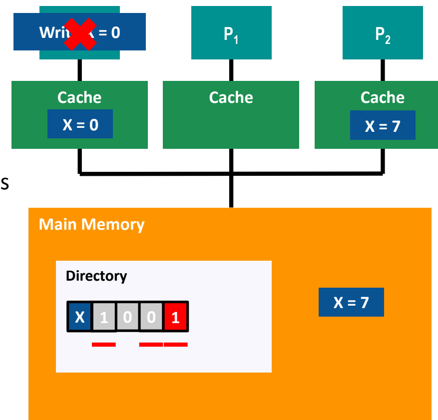
Directory-based CC: Read miss

- P_0 intends to read, misses
- If dirty bit is on
  - Recall cache line from P_j (determine by presence[])
  - Update memory
  - Unset dirty bit, block is shared
  - Set presence[i]
  - Supply data to reader



Directory-based CC: Write miss

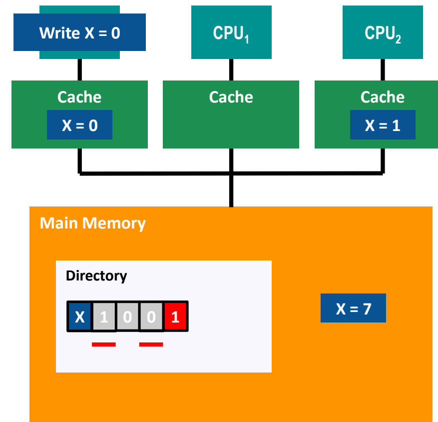
- P_0 intends to write, misses
- If dirty bit (in directory) is off
  - Send invalidations to all P_j with presence[j] turned on
  - Unset presence bit for all processors
  - Set dirty bit
  - Set presence[i], owner P_i



© Markus Püschel
Computer Science
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Directory-based CC: Write miss

- **P_0 intends to write, misses**
- **If dirty bit is on**
  - Recall cache line from owner P_j, invalidate there
  - Write to cache
  - Unset presence[j]
  - Set presence[i]
  - Dirty bit stays on
  - Acknowledge to writer



Read hit and Write hit

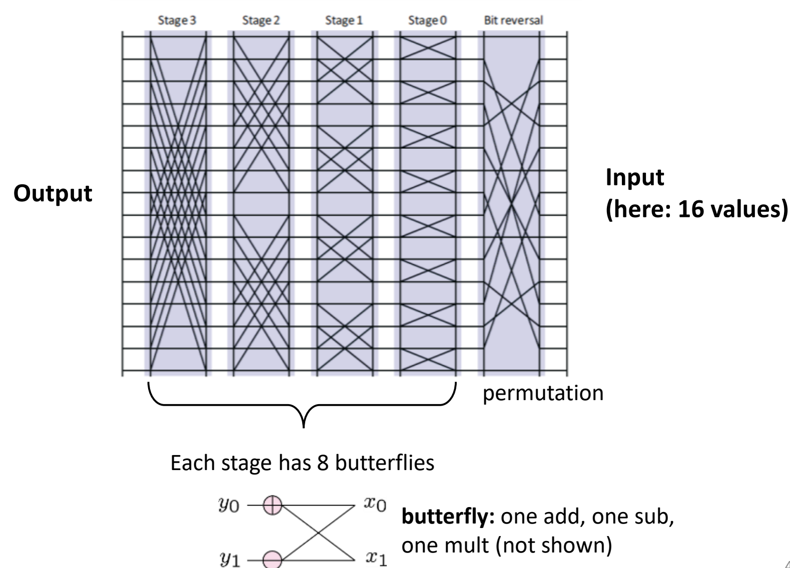
- **Not shown, think about it at home**

© Markus Püschel
Computer Science

Discussion

- **Scaling of memory bandwidth**
  - No centralized memory
- **Directory-based approaches scale with restrictions**
  - Require presence bit for each cache
  - Number of bits determined at design time
  - Directory requires memory (size scales linearly)
  - Shared vs. distributed directory
- **Software emulation**
  - Distributed shared memory (DSM)
  - Emulate cache coherence in software (e.g., TreadMarks)

Example Fast Fourier Transform

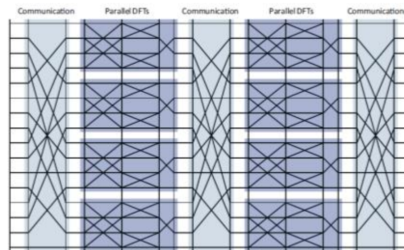


© Markus Püschel
Computer Science
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Example: Fast Fourier Transform

Six-step FFT

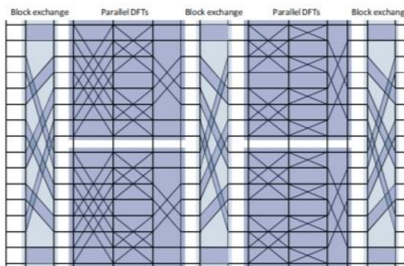
Output



Input (16 values)

Assume: Cache line can hold 2 values

Multi-core FFT



Open Problems (for projects, theses, research)

- **Tune algorithms to cache-coherence schemes**
  - What is the optimal parallel algorithm for a given scheme?
  - Parameterize for an architecture
- **Measure and classify hardware**
  - Read Maranget et al. “A Tutorial Introduction to the ARM and POWER Relaxed Memory Models” and have fun!
  - RDMA consistency is not well understood!
  - GPU memories are not well understood!

    Huge potential for new insights!
- **Can we program (easily) without cache coherence?**
  - How to fix the problems with inconsistent values?
  - Compiler support (issues with arrays)?