Design of Parallel and High-Performance Computing

Fall 2013

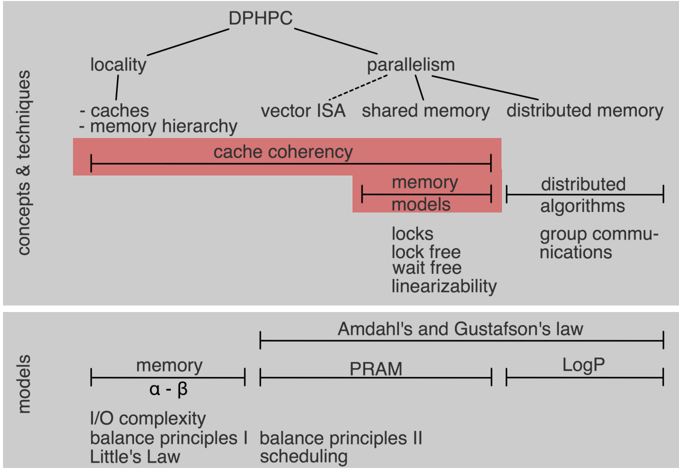
Lecture: Cache Coherence & Memory Models

Instructor: Torsten Hoefler & Markus Püschel

TA: Timo Schneider

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

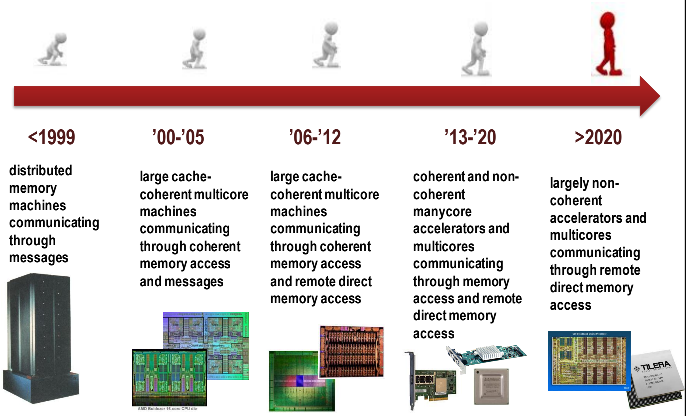
DPHPC Overview



Goals of this lecture

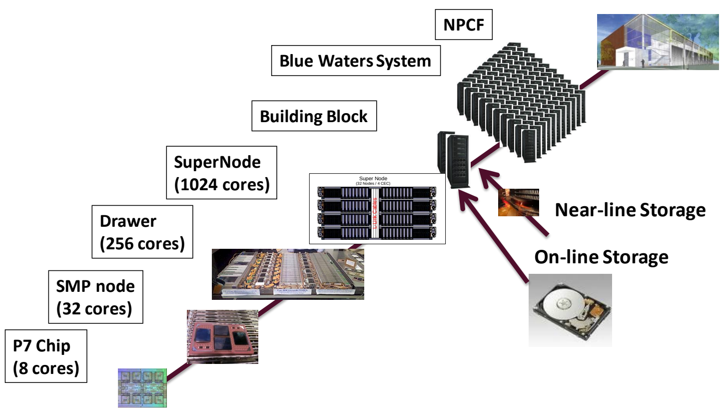
- **Architecture case studies**
- **Memory**
- **Cache Coherence**
- **Memory Consistency**

Architecture Developments



Sources: various vendors

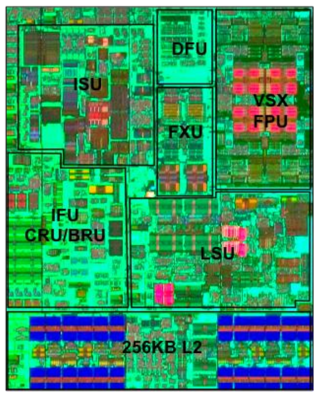
Case Study 1: IBM POWER7 IH (BW)



Source: IBM/NCSA

POWER7 Core

- Execution Units
  - 2 Fixed point units
  - 2 Load store units
  - 4 Double precision floating point
  - 1 Branch
  - 1 Condition register
  - 1 Vector unit
  - 1 Decimal floating point unit
  - 6 wide dispatch
- Recovery Function Distributed
- 1,2,4 Way SMT Support
- Out of Order Execution
- 32KB I-Cache
- 32KB D-Cache
- 256KB L2
  - Tightly coupled to core

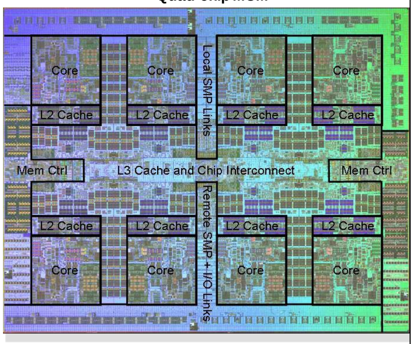


Source: IBM/NCSA

POWER7 Chip (8 cores)

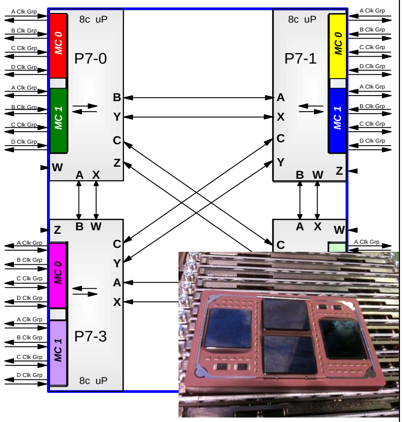
- **Base Technology**
  - 45 nm, 576 mm^2
  - 1.2 B transistors
- **Chip**
  - 8 cores
  - 4 FMAs/cycle/core
  - 32 MB L3 (private/shared)
  - Dual DDR3 memory
    128 GiB/s peak bandwidth
    (1/2 byte/flop)
  - Clock range of 3.5 – 4 GHz

Quad-chip MCM



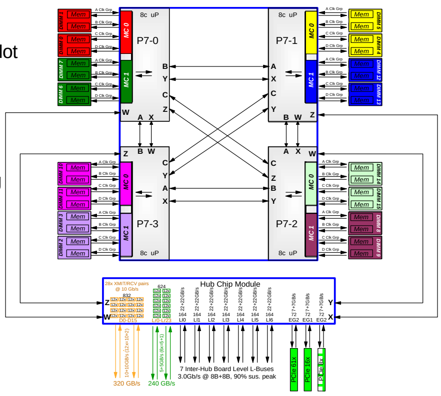
Quad Chip Module (4 chips)

- **32 cores**
  - 32 cores*8 F/core*4 GHz = 1 TF
- **4 threads per core (max)**
  - 128 threads per package
- **4x32 MiB L3 cache**
  - 512 GB/s RAM BW (0.5 B/F)
- **800 W (0.8 W/F)**



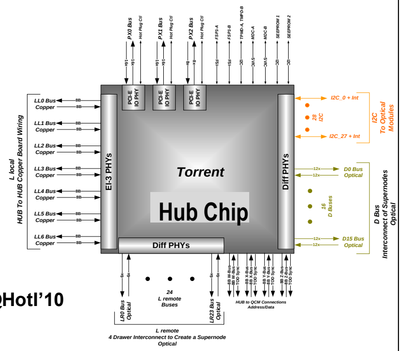
Adding a Network Interface (Hub)

- **Connects QCM to PCI-e**
  - Two 16x and one 8x PCI-e slot
- **Connects 8 QCM's via low latency, high bandwidth, copper fabric.**
  - Provides a message passing mechanism with very high bandwidth
  - Provides the lowest possible latency between 8 QCM's



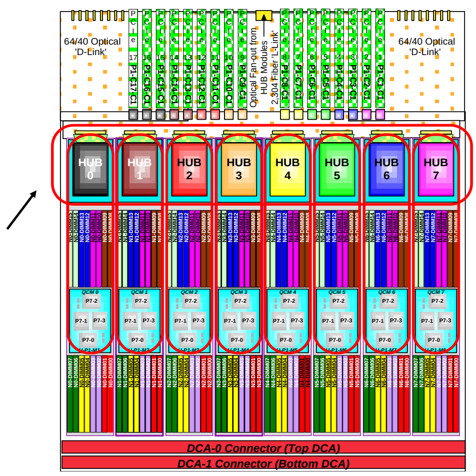
1.1 TB/s POWER7 IH HUB

- 192 GB/s Host Connection
- 336 GB/s to 7 other local nodes
- 240 GB/s to local-remote nodes
- 320 GB/s to remote nodes
- 40 GB/s to general purpose I/O
- cf. "The PERCS interconnect" @Hotl'10



P7 IH Drawer

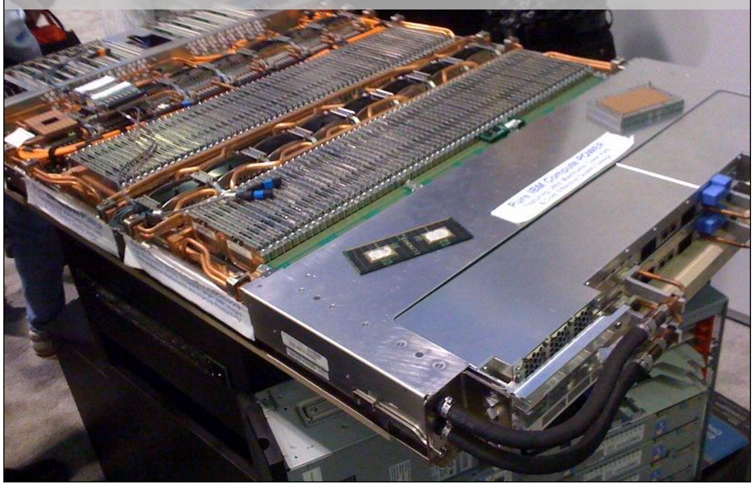
- 8 nodes
  - 32 chips
  - 256 cores



First Level Interconnect

- L-Local
- HUB to HUB Copper Wiring
- 256 Cores

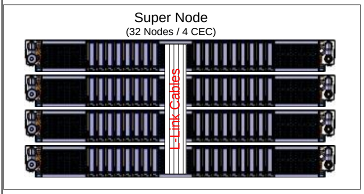
POWER7 IH Drawer @ SC09

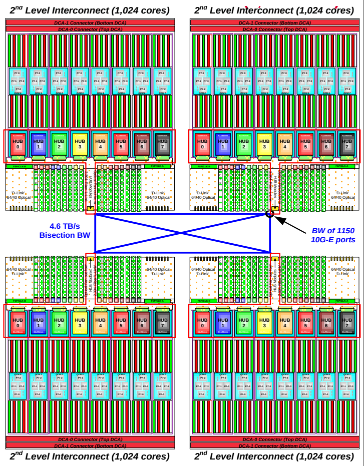


P7 IH Supernode

Second Level Interconnect

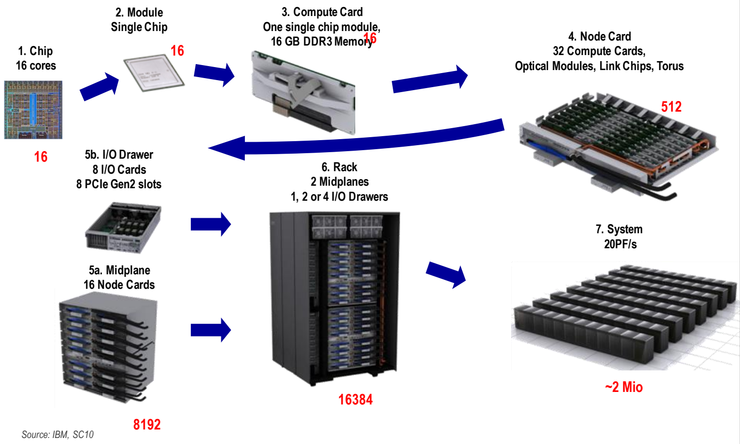
- Optical 'L-Remote' Links from HUB
- 4 drawers
- 1,024 Cores





Source: IBM/NCSA

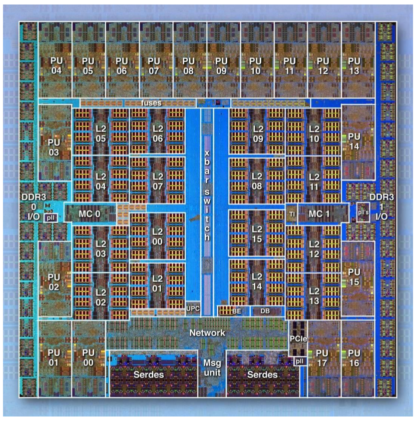
Case Study 2: IBM Blue Gene/Q packaging



Source: IBM, SC10

Blue Gene/Q Compute chip

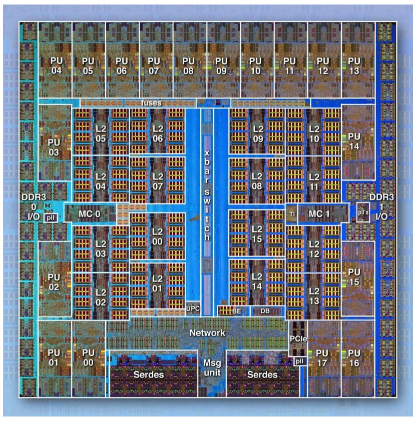
System-on-a-Chip design : integrates processors, memory and networking logic into a single chip



- **360 mm² Cu-45 technology (SOI)**
  - ~ 1.47 B transistors
- **16 user + 1 service processors**
  - plus 1 redundant processor
  - all processors are symmetric
  - each 4-way multi-threaded
  - 64 bits PowerISA™
  - 1.6 GHz
  - L1 I/D cache = 16kB/16kB
  - L1 prefetch engines
  - each processor has Quad FPU (4-wide double precision, SIMD)
  - peak performance 204.8 GFLOPS@55W
- **Central shared L2 cache: 32 MB**
  - eDRAM
  - multiversioned cache/transactional memory/speculative execution.
  - supports atomic ops
- **Dual memory controller**
  - 16 GB external DDR3 memory
  - 1.33 GHz
  - 2 * 16 byte-wide interface (+ECC)
- **Chip-to-chip networking**
  - Router logic integrated into BQC chip.

Source: IBM, PACT'11

Blue Gene/Q Network

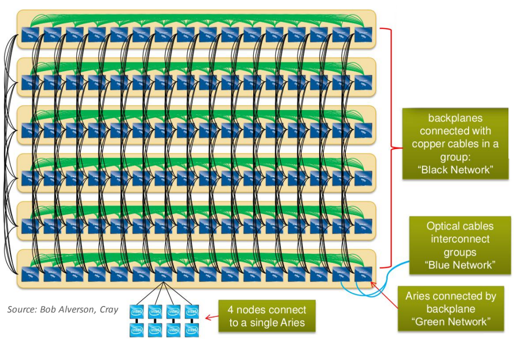


- **On-chip external network**
  - Message Unit
  - Torus Switch
  - Serdes
  - Everything!
- **Only 55-60 W per node**
  - Top of Green500 and GreenGraph500

Source: IBM, PACT'11

Case Study 3: Cray Cascade (XC30)

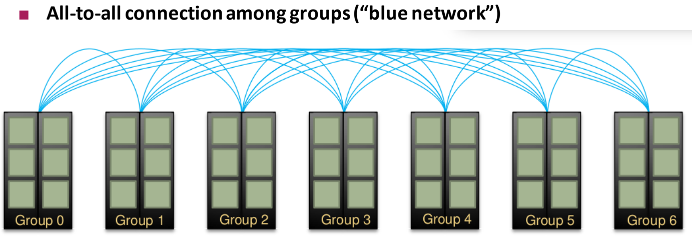
- **Biggest current installation at CSCS! ☺**
  - >2k nodes
- **Standard Intel x86 Sandy Bridge Server-class CPUs**



Source: Bob Alverson, Cray

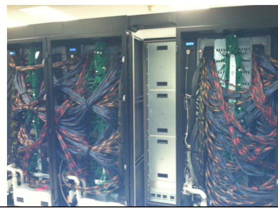
Cray Cascade Network Topology

- **All-to-all connection among groups ("blue network")**



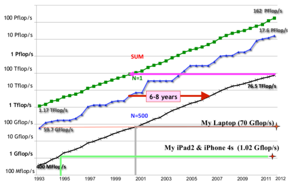
Source: Bob Alverson, Cray

- **What does that remind you of?**



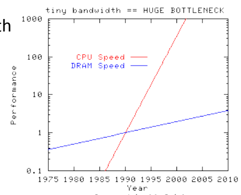
Memory – CPU gap widens

- **Measure processor speed as “throughput”**
  - FLOPS/s, IOPS/s, …
  - Moore’s law - ~60% growth per year



Source: Jack Dongarra

- **Today’s architectures**
  - POWER7: 256 GFLOP/s – 128 GB/s memory bandwidth
  - BG/Q: 205 GFLOPS/s – 42.6 GB/s memory bandwidth
  - Trend: memory performance grows 10% per year



Source: John Mc.Calpin

Issues

- **How to measure bandwidth?**
  - Data sheet (often peak performance, may include overheads)
    Frequency times bus width: 51 GiB/s
  - Microbenchmark performance
    Stride 1 access (32 MiB): 32 GiB/s
    Random access (8 B out of 32 MiB): 241 MiB/s
    Why?
  - Application performance
    As observed (performance counters)
    Somewhere in between stride 1 and random access
- **How to measure Latency?**

Issues

- **How to measure bandwidth?**
  - Data sheet (often peak performance, may include overheads)
    Frequency times buswidth: 51 GiB/s
  - Microbenchmark performance
    Stride 1 access (32 MiB): 32 GiB/s
    Random access (8 B out of 32 MiB): 241 MiB/s
    Why?
  - Application performance
    As observed (performance counters)
    Somewhere in between stride 1 and random access
- **How to measure Latency?**
  - Data sheet (often optimistic, or not provided)
    <100ns
  - Random pointer chase
    110 ns with one core, 258 ns with 32 cores!

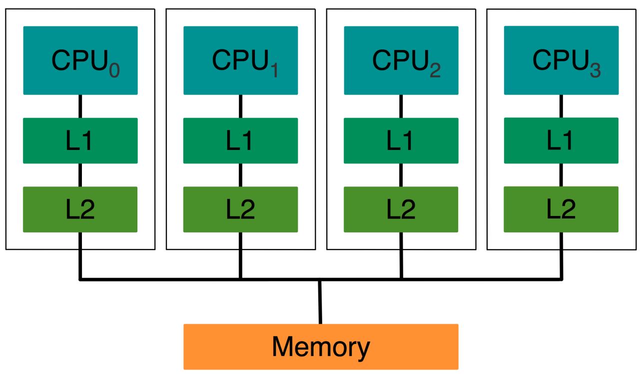
Conjecture: Buffering is a must!

- **Write Buffers**
  - Delayed write back saves memory bandwidth
  - Data is often overwritten or re-read
- **Caching**
  - Directory of recently used locations
  - Stored as blocks (cache lines)

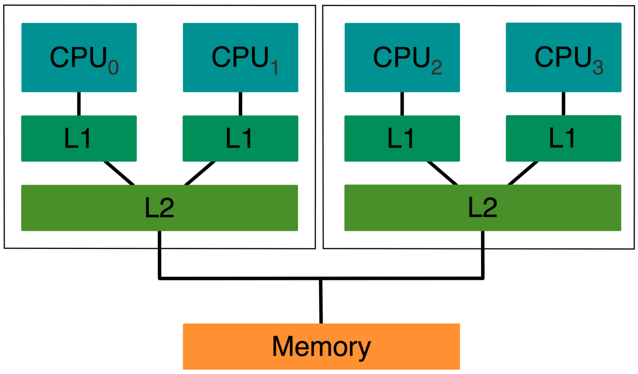
Cache Coherence

- **Different caches may have copy if same memory location!**
- **Cache coherence**
  - Manages existence of multiple copies
- **Cache architectures**
  - Multi level caches
  - Multi-port vs. single port
  - Shared vs. private (partitioned)
  - Inclusive vs. exclusive
  - Write back vs. write through
  - Victim cache to reduce conflict misses
  - …

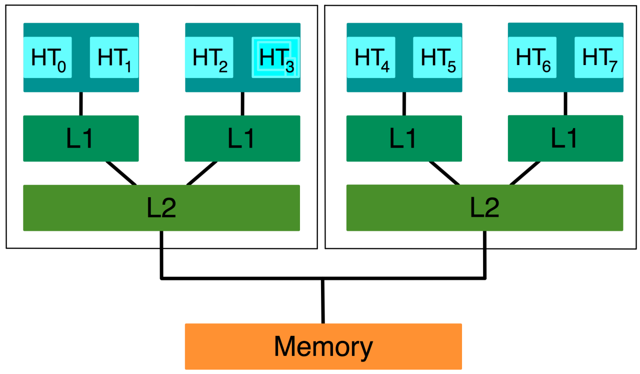
Exclusive Hierarchical Caches



Shared Hierarchical Caches



Shared Hierarchical Caches with MT



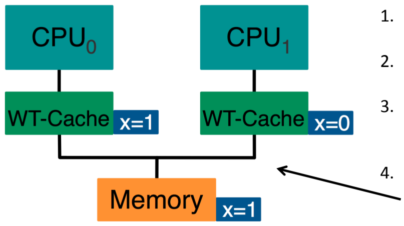
Caching Strategies (repeat)

- **Remember:**
  - Write Back?
  - Write Through?
- **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)

Write Through Cache



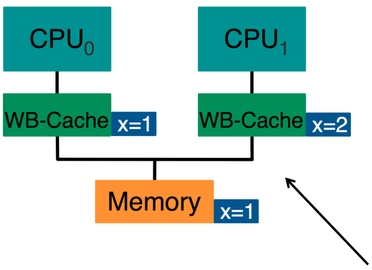
1. CPU_0 reads X from memory
   - loads X=0 into its cache
2. CPU_1 reads X from memory
   - loads X=0 into its cache
3. CPU_0 writes X=1
   - stores X=1 in its cache
   - stores X=1 in memory
4. CPU_1 reads X from its cache
   - loads X=0 from its cache

   Incoherent value for X on CPU_1

   CPU_1 may wait for update!

Requires write propagation!

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!

A simple example

- **Assume C99:**

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

- **Two threads:**
  - Thread 0: write to a
  - Thread 1: write to b
- **Assume write back cache**
  - What may end up in main memory?

Cache Coherence Protocol

- **Programmer cannot deal with unpredictable behavior!**
- **Cache controller maintains data integrity**
  - All writes to different locations are visible

Fundamental Mechanisms

- **Snooping**
  - Shared bus or (broadcast) network
  - Cache controller "snoops" all transactions
  - Monitors and changes the state of the cache's data
- **Directory-based**
  - Record information necessary to maintain cohrence
  - E.g., owner and state of a line etc.

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)
  - Precision of block sharing (per block, per sub-block?)
  - Block size (cache line size?)

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

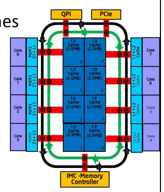
Cache Coherence Approaches

- **Based on invalidation**
  - Broadcast all coherency traffic (writes to shared lines) to all caches
  - Each cache snoops

    Invalidate lines written by other CPUs

    Signal sharing for cache lines in local cache to other caches
  - Simple implementation for bus-based systems
  - Works at small scale, challenging at large-scale

    E.g., Intel Sandy Bridge



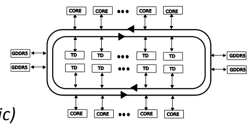
Source: Intel

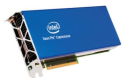
- **Based on explicit updates**
  - Central directory for cache line ownership
  - Local write updates copies in remote caches

    Can update all CPUs at once

    Multiple writes cause multiple updates (more traffic)
  - Scalable but more complex/expensive

    E.g., Intel Xeon Phi





Invalidation vs. update

- **Invalidation-based:**
  - Only write misses hit the bus (works with write-back caches)
  - Subsequent writes to the same cache line are local
  - → Good for multiple writes to the same line (in the same cache)
- **Update-based:**
  - All sharers continue to hit cache line after one core writes

    Implicit assumption: shared lines are accessed often
  - Supports producer-consumer pattern well
  - Many (local) writes may waste bandwidth!
- **Hybrid forms are possible!**

MESI Cache Coherence

- **Most common hardware implementation of discussed requirements**

  aka. "Illinois protocol"

Each line has one of the following states (in a cache):

- **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

Terminology

- **Clean line:**
  - Content of cache line and main memory is identical (also: memory is up to date)
  - Can be evicted without write-back
- **Dirty line:**
  - Content of cache line and main memory differ (also: 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
- **Local 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
- **State is E**
  - No bus transaction
- **State is S**
  - No bus transaction
- **State is I**
  - Generate bus read request (BusRd)

    May force other cache operations (see later)
  - Other cache(s) signal "sharing" if they hold a copy
  - If shared was signaled, go to state S
  - Otherwise, go to state E
- **After update: return read value**

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 request for upgrade to exclusive (BusRdX*)
  - Go to state M
- **State is I**
  - Generate bus read request for exclusive ownership (BusRdX)
  - Go to state M

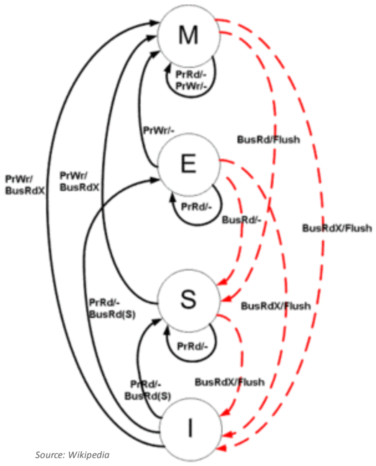
Transitions in response to snooped BusRd

- **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 and signal "shared"
- **State is S**
  - Signal "shared"
- **State is I**
  - Ignore

Transitions in response to snooped BusRdX

- **State is M**
  - Write cache line back to memory
  - Discard line and go to I
- **State is E**
  - Discard line and go to I
- **State is S**
  - Discard line and go to I
- **State is I**
  - Ignore
- **BusRdX* is handled like BusRdX!**

MESI State Diagram (FSM)



Source: Wikipedia

Small Exercise

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

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
P1 writes x	M	I	I	BusRdX*	Cache
P1 reads x	M	I	I	-	Cache
P3 writes x	I	I	M	BusRdX	Memory

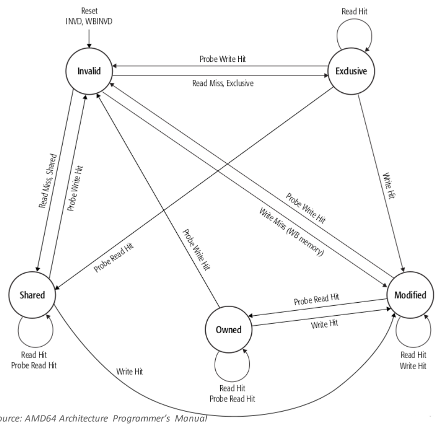
Optimizations?

- **Class question: what could be optimized in the MESI protocol to make a system faster?**

Related Protocols: MOESI (AMD)

- **Extended MESI protocol**
- **Cache-to-cache transfer of modified cache lines**
  - Cache in M or 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!
- **Broadcasts updates in O state**
  - Additional load on the bus

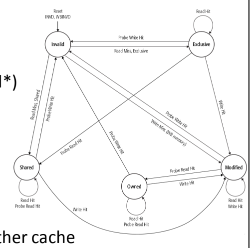
MOESI State Diagram



Source: AMD64 Architecture Programmer's Manual

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



Related Protocols: MESIF (Intel?)

- **Modified (M): Modified Exclusive**
  - No copies in other caches, local copy dirty
  - 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
- **Forward (F):**
  - Special form of S state, other caches may have line in S
  - Most recent requester of line is in F state
  - Cache acts as responder for requests to this line

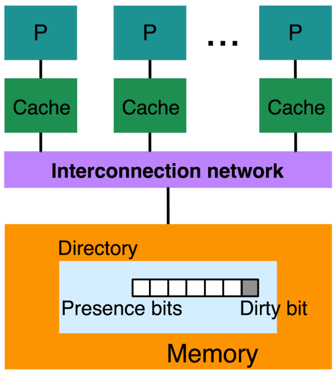
Multi-level caches

- **Most systems have multi-level caches**
  - Problem: only “last level cache” is connected to bus or network
  - 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
- **If L1 is write through, L2 could “remember” state of L1 cache line**
  - May increase traffic though

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 (entry for each CL) 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



- System with N processors P_i
- For each memory block (size: cache line) maintain a directory entry
  - N presence bits
  - Set if block in cache of P_i
  - 1 dirty bit
- For each cache block
  - 1 valid and 1 dirty bit
- First proposed by Censier and Feautrier (1978)

Directory-based CC: Read miss

- **P_i intends to read, misses**
- **If dirty bit (in directory) is off**
  - Read from main memory
  - Set presence[i]
  - Supply data to reader
- **If dirty bit is on**
  - Recall cache line from P_j
  - Update memory
  - Unset dirty bit, block shared
  - Set presence[i]
  - Supply data to reader

Directory-based CC: Write miss

- **P_i intends to write, misses**
- **If dirty bit (in directory) is off**
  - Send invalidations to all processors P_j with presence[j] turned on
  - Unset presence bit for all processors
  - Set dirty bit
  - Set presence[i], owner P_i
- **If dirty bit is on**
  - Recall cache line from owner P_j
  - Update memory
  - Unset presence[j]
  - Set presence[i], dirty bit remains set
  - Supply data to reader

Directory-based CC: Write hit on remote

- P_i **intends to write, misses**

- **Cache line valid, dirty bit off,** P_i **not owner**
  - Access directory
  - Send invalidations to all processors P_j with presence[j] set
  - Unset presence bit for all processors
  - Set dirty bit
  - Set presence[i], owner P_i

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)
  - Often on a per-page basis, utilizes memory virtualization and paging

Case Study: Intel Xeon Phi