TECHNISCHE UNIVERSITÄT DRESDEN

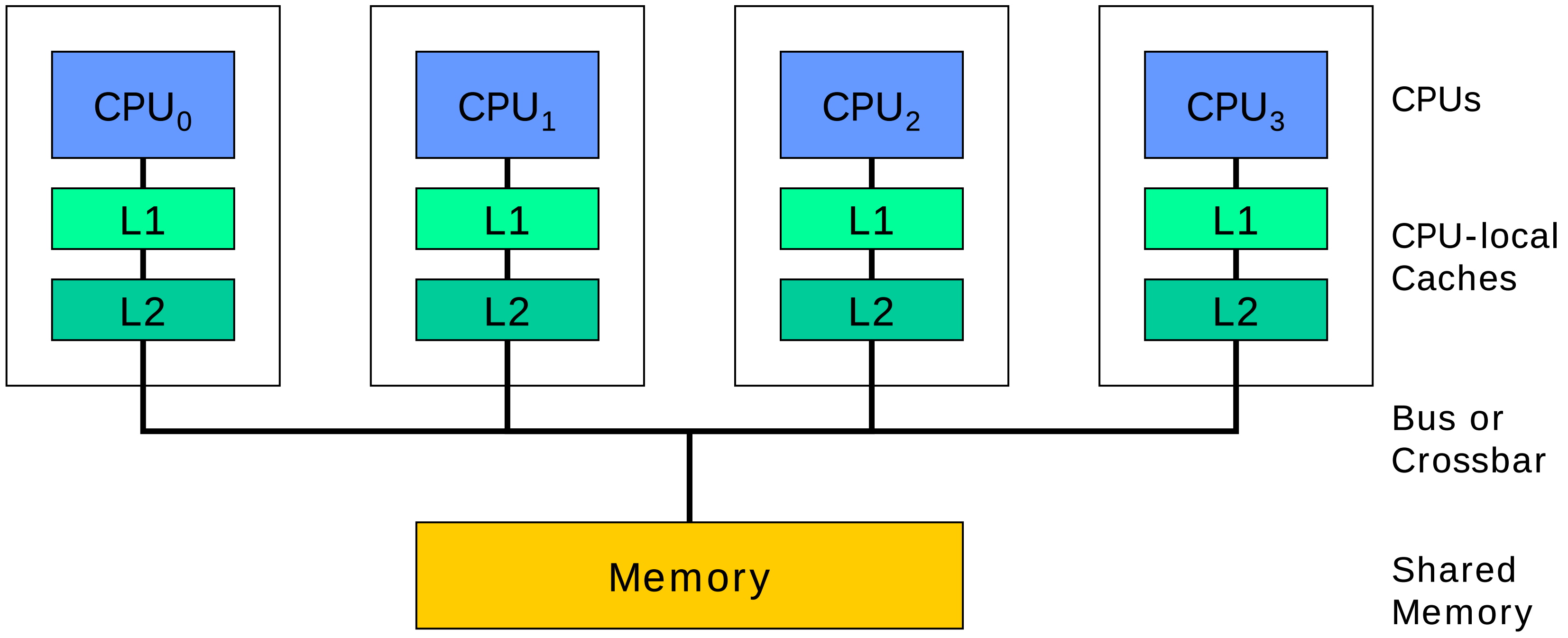
Department of Computer Science - Institute for System Architecture, Operating Systems Group

Parallel Architectures

Memory Consistency & Cache Coherency

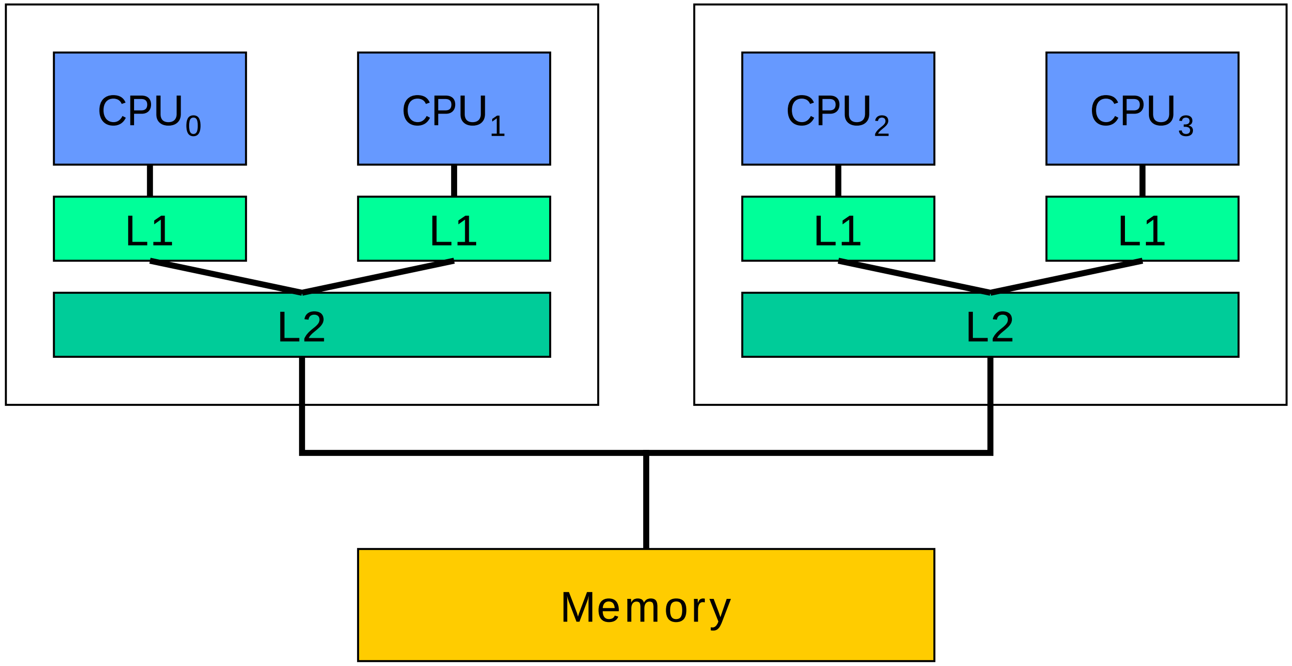
Udo Steinberg

Symmetric Multi-Processor (SMP)



TECHNISCHE UNIVERSITÄT DRESDEN

Chip Multi-Processor (CMP), Multicore



CPUs

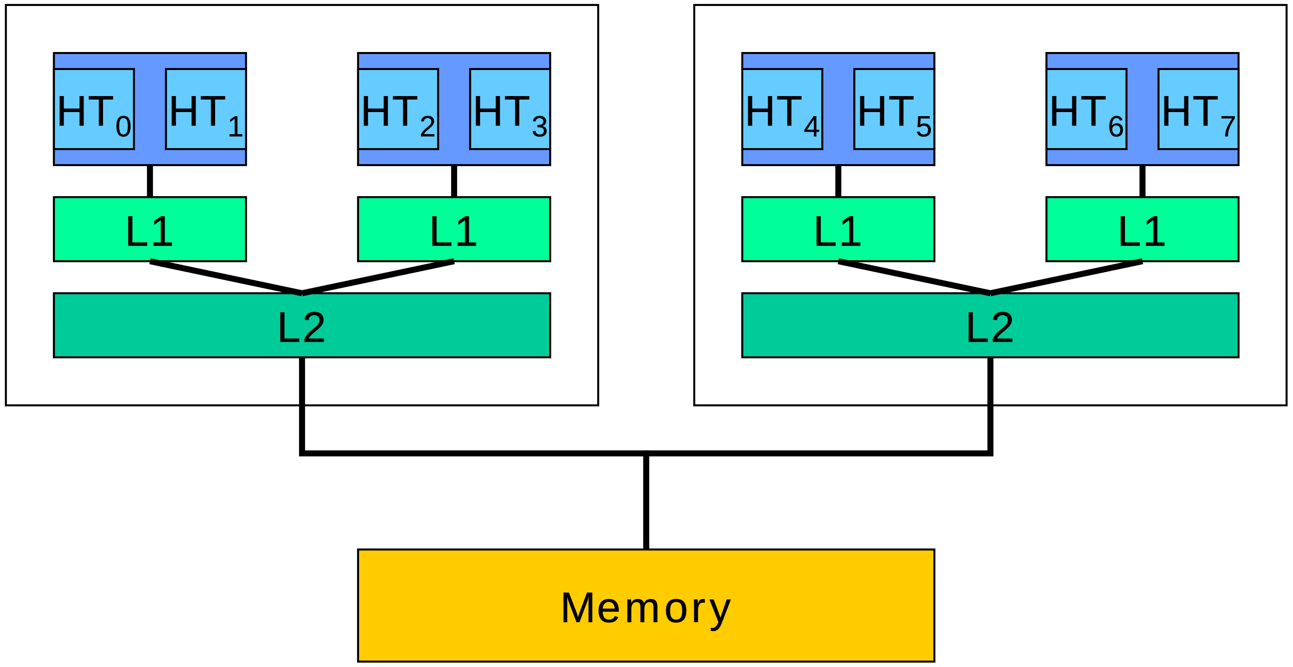
CPU-local Cache

Shared Cache

Bus or Crossbar

Shared Memory

Symmetric Multi-Threading (SMT), Hyperthreading



Logical Processors

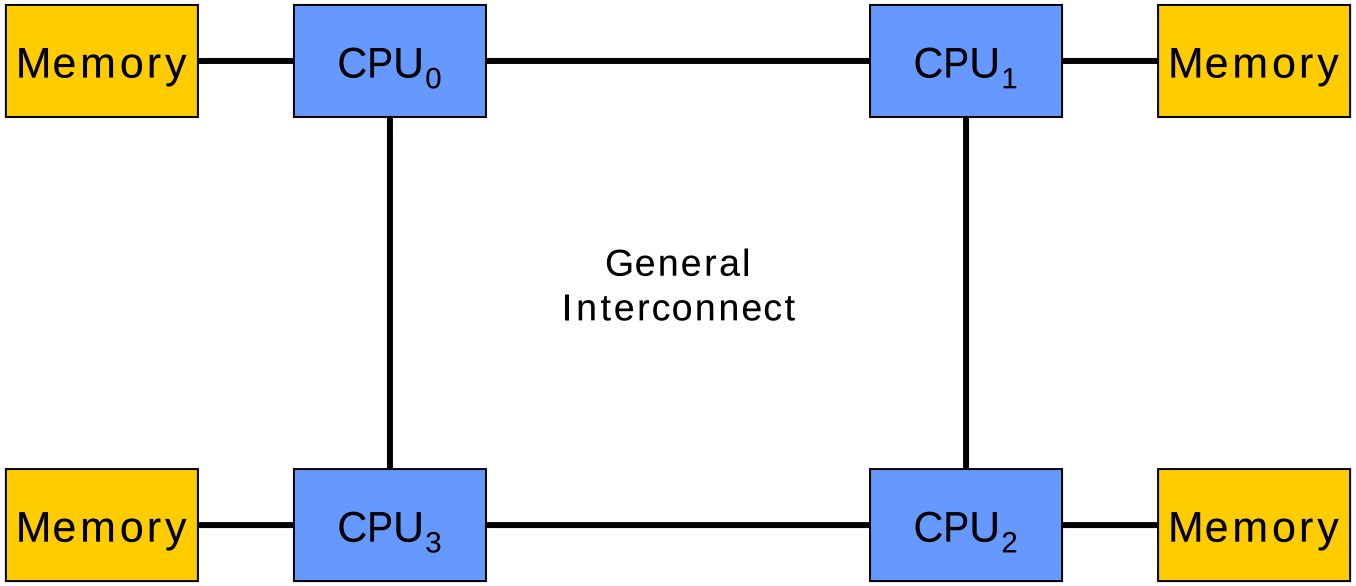
CPU-local Cache

Shared Cache

Bus or Crossbar

Shared Memory

Non-Uniform Memory Access (NUMA)



Multi-Processor Systems and Shared Memory

- Multiple processors share memory

- Memory managed by one or more memory controllers
  - UMA (Uniform Memory Access)
  - NUMA (Non-Uniform Memory Access)

- What is memory behavior under concurrent data access?
  - Reading a memory location should return last value written
  - „Last value written“ not clearly defined under concurrent access

- Defined by system‘s memory consistency model
  - Defines the order that processors perceive concurrent accesses
  - Based on ordering, not timing of accesses

Memory Consistency Models

- Different memory consistency models exist
  - Some platforms (e.g., SPARC) support multiple models

- More complex models attempt to expose more performance

- Terminology:
  - Program Order (of a processor‘s operations)
    - per-processor order of memory accesses determined by program (software)

  - Visibility Order (of all operations)
    - order of memory accesses observed by one or more processors
    - every read from a location returns value of most recent write

Most Intuitive Model: Sequential Consistency

- A multiprocessor system is <u>sequentially consistent</u> if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. (Lamport 1979)

- <u>Program Order Requirement</u>
  - each CPU issues memory operations in program order

- <u>Atomicity Requirement</u>
  - Memory services operations one-at-a-time
  - all memory operations appear to execute atomically with respect to other memory operations

Examples for Sequential Consistency

CPU_1		**CPU_2**	
A = 1;	(a_1)	u = B;	(a_2)
B = 1;	(b_1)	v = A;	(b_2)

(u,v) = (1,1) sequentially consistent

» example visibility order: a_1,b_1,a_2,b_2

(u,v) = (1,0) sequentially inconsistent

» example visibility order: b_1,a_2,b_2,a_1

This visibility order violates program order on CPU_1

No visibility order exists that satisfies program order on all CPUs and produces (u,v) = (1,0) result

TECHNISCHE UNIVERSITÄT DRESDEN

Examples for Sequential Consistency

CPU$_1$		**CPU$_2$**	
A = 1;	(a_1)	B = 1;	(a_2)
u = B;	(b_1)	v = A;	(b_2)

(u,v) = (1,1) sequentially consistent

» example visibility order: a_1,a_2,b_1,b_2

(u,v) = (0,0) sequentially inconsistent

» example visibility order: b_1,b_2,a_1,a_2

This visibility order violates program order on CPU$_{1/2}$

No visibility order exists that satisfies program order on all CPUs and produces (u,v) = (0,0) result

Sequential Consistency vs. Architecture Optimizations

CPU$_1$		**CPU$_2$**	
A = 1;	(a_1)	u = B;	(a_2)
B = 1;	(b_1)	v = A;	(b_2)

- Relaxing the Program Order:
  - Out-of-order execution may reorder operations (b_2,a_2)
  - Store Buffer may reorder writes (b_1,a_1)
  - produces sequentially inconsistent result (u,v) = (1,0)

- Maintaining Program Order:
  - May still produce sequentially inconsistent results
    - CPU$_1$ issues a_1,b_1 in program order
    - but a_1 misses and b_1 hits in the cache (non-blocking cache)

Sequential Consistency vs. Architecture Optimizations

CPU_1	**CPU_2**	**CPU_3**

A = 1;	while (A == 0);
	B = 1;
		while (B == 0);
		print A;

A in cache	*A in cache*	*A, B in cache*

- Relaxing the Atomicity of Writes:
  1. CPU_1 writes A = 1, generates update message to CPU_2 and CPU_3
  2. CPU_2 receives update message for A from CPU_1
  3. CPU_2 writes B = 1, generates update message to CPU_3
  4. CPU_3 receives update message for B from CPU_2
  5. CPU_3 prints A = 0
  6. CPU_3 receives update message for A from CPU_1
  - Sequentially inconsistent result, because write to A not atomic wrt. other memory operations (e.g., write to B)

TECHNISCHE UNIVERSITÄT DRESDEN

Sequential Consistency vs. Compiler Optimizations

Programmer‘s Code

CPU_1	CPU_2
A = 1;	while (Flag == 0);
Flag = 1;	u = A;

Generated Code

CPU_1	CPU_2
A = 1;	reg = Flag;
Flag = 1;	while (reg == 0);
	u = A;

- Compiler optimizations such as
  - register allocation and value caching
  - code motion
  - common sub-expression elimination
  - loop interchange

  can reorder memory operations similar to architecture optimizations or even eliminate memory operations completely

Relaxing Write-to-Read or Write-to-Write Order

- Write-to-Read (later reads can bypass earlier writes):
  - Write followed by a read can execute out-of-order
  - Typical hardware usage: Store Buffer
    - Writes must wait for ownership of cache line
    - Reads can bypass writes in the store buffer
    - Hides write latency

- Write-to-Write (later writes can bypass earlier writes) :
  - Write followed by another write can execute out-of-order
  - Typical hardware usage: non-blocking cache, write coalescing
    - Writes must wait for ownership of cache line
    - Latency for obtaining ownership depends on hop count to cache line owner

IBM-370 (zSeries)

- In-order memory operations:
  - Read-to-Read
  - Read-to-Write
  - Write-to-Write

- Out-of-order memory operations:
  - Write-to-Read (later reads can bypass earlier writes)
    - unless both are to the same memory location
    - breaks Dekker‘s algorithm for mutual exclusion

  - Write-to-Read to same location must execute in-order
    - no forwarding of pending writes from the store buffer

Dekker‘s Algorithm on IBM-370 (zSeries)

```
bool flag0 = false, flag1 = false;
int turn = 0;
```

CPU #0

```
P: flag0 = true;          // Buffered
   while (flag1) {
      if (turn == 1) {
         flag0 = false;
         goto P;
      }
   }
   // critical section
   flag0 = false;
   turn = 1;
```

CPU #1

```
P: flag1 = true;          // Buffered
   while (flag0) {
      if (turn == 0) {
         flag1 = false;
         goto P;
      }
   }
   // critical section
   flag1 = false;
   turn = 0;
```

SPARC V8 Total Store Order (TSO)

- In-order memory operations:
  - Read-to-Read
  - Read-to-Write
  - Write-to-Write

- Out-of-order memory operations:
  - Write-to-Read (later reads can bypass earlier writes)
    - Forwarding of pending writes in the store buffer to successive read operations of the same location
      - Writes become visible to writing processor first

    - Breaks Peterson‘s algorithm for mutual exclusion

Peterson‘s Algorithm on SPARC V8 TSO

```
bool flag0 = false, flag1 = false;
int turn = 0;
```

CPU #0

```
flag0 = true;
turn = 1;
while (turn == 1 && flag1) {};
// critical section
flag0 = false;
```

CPU #1

```
flag1 = true;
turn = 0;
while (turn == 0 && flag0) {};
// critical section
flag1 = false;
```

TECHNISCHE UNIVERSITÄT DRESDEN

Total Store Order (TSO) vs. SC and IBM-370

CPU$_1$		**CPU$_2$**	
A = 1;	(a_1)	B = 1;	(a_2)
u = A;	(b_1)	v = B;	(b_2)
w = B;	(c_1)	x = A;	(c_2)

- (u,v,w,x) = (1,1,0,0)
  - not possible with Sequential Consistency (SC) and IBM-370
  - but possible with Total Store Order (TSO)
    - Example total order: b_1,b_2,c_1,c_2,a_1,a_2
    - b_1 reads A=1 from store buffer
    - b_2 reads B=1 from store buffer
    - A=1 globally visible after c_2
    - B=1 globally visible after c_1

Processor Consistency (PC)

- Similar to Total Store Order (TSO)

- But model additionally supports multiple cached memory copies
  - Relaxed atomicity for write operations
    - Each write operation broken into sub-operations to update cached copies of other CPUs
  - Non-unique write order, requires per-CPU visibility order

  - Additional Coherency Requirement:
    - All writes sub-operations to the same memory location complete in the same order across all memory copies (or in other words: every processor should see writes to the **same** location in the same order)
    - If one CPU observes writes to X in the order $W_1(X)$ before $W_2(X)$, another CPU must not see $W_2(X)$ before $W_1(X)$

TECHNISCHE UNIVERSITÄT DRESDEN

Processor Consistency (PC) vs. SC, IBM-370 and TSO

CPU_1	**CPU_2**	**CPU_3**
A = 1; (a_1)	u = A; (a_2)	v = B; (a_3)
	B = 1; (b_2)	w = A; (b_3)

- (u,v,w) = (1,1,0)
  - not possible with SC, IBM-370 and TSO
  - but possible with Processor Consistency (PC)
    - CPU_1 sets A = 1, sends $W_1(A)$ to other CPUs
    - CPU_2 observes $W_1(A)$, sets B = 1, sends $W_2(B)$ to other CPUs
    - CPU_3 observes $W_2(B)$ ... but has not yet received $W_1(A)$
  - Single memory bus enforces single visibility order
  - Multiple visibility orders possible with other topologies

SPARC V8 Partial Store Order (PSO)

- In-order memory operations:
  - Read-to-Read
  - Read-to-Write
- Out-of-order memory operations:
  - Write-to-Read (later reads can bypass earlier writes)
    - Forwarding of pending writes in the store buffer to successive read operations of the same location
  - Write-to-Write (later writes can bypass earlier writes)
    - unless both are to the same memory location
    - breaks Producer-Consumer Code
- Write Atomicity is maintained -> single visibility order

Partial Store Order (PSO) vs. SC, IBM-370, TSO and PC

CPU$_1$		**CPU$_2$**	
A = 1;	(a_1)	while (Flag == 0);	(a_2)
B = 1;	(b_1)	u = A;	(b_2)
Flag = 1;	(c_1)	v = B;	(c_2)

- (u,v) = (0,0) or (0,1) or (1,0)
  - not possible with SC, IBM-370, TSO and PC
  - but possible with Partial Store Order (PSO)
    - Example total order: c_1,a_2,b_2,c_2,b_1,a_1
    - Store barrier (STBAR) before c_1 ensures sequentially consistent result (u,v) = (1,1)

Relaxing all Program Orders

- In addition to previous relaxations:
  - Read-to-Read (later reads can bypass earlier reads) :
    - Read followed by a read can execute out-of-order
  - Read-to-Write (later writes can bypass earlier reads):
    - Read followed by a write can execute out-of-order

- Examples:
  - Weak Ordering (WO)
  - Release Consistency (RC)
  - DEC Alpha
  - SPARC V9 Relaxed Memory Order (RMO)
  - PowerPC
  - Itanium (IA64)

Weak Ordering (WO)

- Conceptually similar to Processor Consistency (PC)
  - including coherency requirement
- Classifies memory operations into two categories:
  - data operations
  - synchronization operations
- Reordering of memory accesses between synchronization operations typically does not affect correctness of a program
- Program order only maintained at synchronization points
  - between synchronization operations

Release Consistency (RC)

- Distinguishes memory operations as
  - ordinary (data)
  - special
    - sync (synchronization)
    - nsync (asynchronous data)

- Sync operations classified as
  - acquire
    - read operation for gaining access to a shared resource
    - e.g., spinning on a flag to be set
  - release
    - write operation for granting permission to a shared resource
    - e.g., setting a synchronization flag

Flavors of Release Consistency (RC)

- RC_{SC}
  - Sequential consistency between special operations
  - Program order enforced between:
    - acquire -> all
    - all -> release
    - special -> special
- RC_{PC}
  - Processor consistency between special operations
  - Program order enforced between:
    - acquire -> all
    - all -> release
    - special -> special
      - except special write followed by special read
      - can use read-modify-write instruction to achieve effect

TECHNISCHE UNIVERSITÄT DRESDEN

Memory Consistency in Modern Architectures

Reordered Memory Accesses	Read-to-Read	Read-to-Write	Write-to-Write	Write-to-Read	Atomic OPs and Reads	Atomic OPs and Writes
Alpha	Y	Y	Y	Y	Y	Y
AMD64	Y			Y		
IA64	Y	Y	Y	Y	Y	Y
PA-RISC	(Y)	(Y)	(Y)	(Y)		
POWER*	Y	Y	Y	Y	Y	Y
SPARC RMO	Y	Y	Y	Y	Y	Y
SPARC PSO			Y	Y		Y
SPARC TSO				Y		
IA32*	Y	Y		Y		

*write atomicity relaxed

Enforcing Ordering: Synchronization Instructions

- IA32/AMD64:
  - lfence (load fence), sfence (store fence), mfence (memory fence)
- Alpha:
  - mb (memory barrier), wmb (write memory barrier)
- SPARC (PSO)
  - stbar (store barrier)
- SPARC (RMO)
  - membar (4-bit encoding for r-r, r-w, w-r, w-w ordering)
- PowerPC
  - sync (similar to Alpha mb, except for r-r), lwsync

TECHNISCHE UNIVERSITÄT DRESDEN

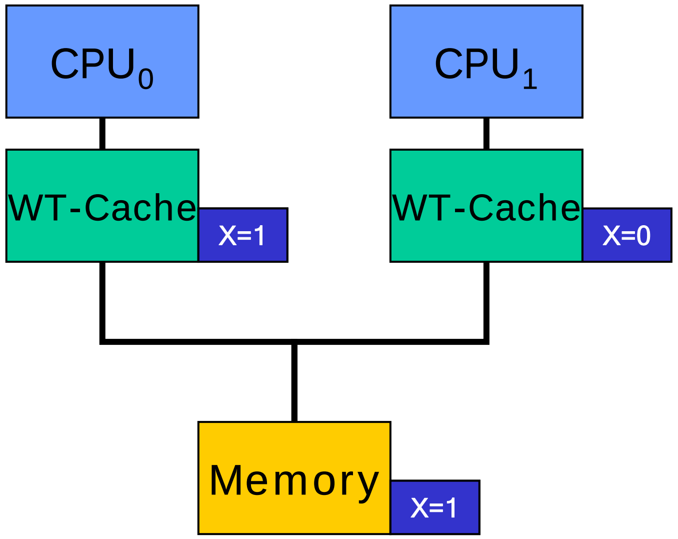
Cache Coherency

- Caching leads to presence of multiple copies for a memory location

- Cache coherency is a mechanism for keeping copies up-to-date
  - locate all cached copies of a memory location
  - eliminate stale copies (invalidate/update)

- Requirements:
  - Write Propagation: Writes must eventually become visible to all processors
  - Write Serialization: Every processor should see the writes to the **same** location in the same order

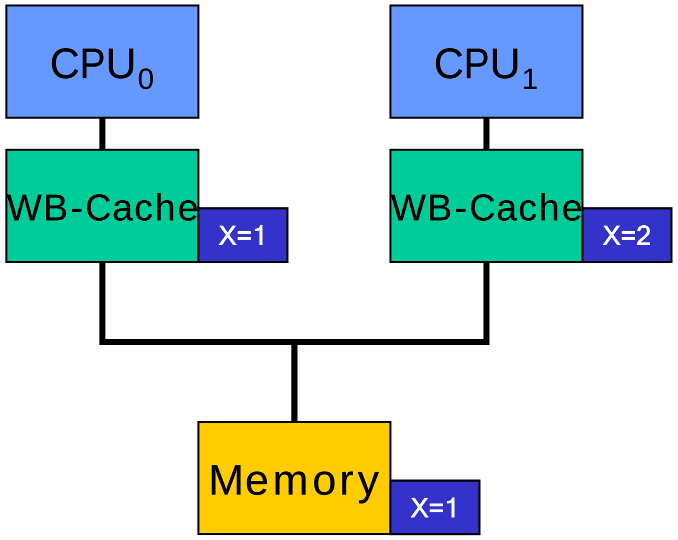
Incoherency Example (1)



1. CPU_0 reads X from memory
   - stores X=0 into its cache
2. CPU_1 reads X from memory
   - stores 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

Incoherency Example (2)



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 memory
6. CPU_0 writes back cache line
   - stores X=1 in memory

Later store X=2 from CPU_1 lost

Cache Coherency: Problems and Solutions

- Problem 1:
  CPU$_1$ used stale value that had already been modified by CPU$_0$
  - Solution:
    Invalidate all copies before allowing a write to proceed

- Problem 2:
  Incorrect writeback order of modified cache lines
  - Solution:
    Disallow more than one modified copy

Coherency Protocol Approaches

- Invalidation-based
  - all coherency-related traffic broadcast to all CPUs
  - each processor snoops traffic and reacts accordingly
    - invalidate lines written to by another CPU
    - signal sharing for cache lines currently in cache
  - straightforward solution for bus-based systems
  - suited for small-scale systems

- Update-based
  - Uses central directory for cache line ownership
  - Write operation updates copies in other caches
    - can update all other CPUs at once (less bus traffic)
    - but: multiple writes cause multiple updates (more bus traffic)
  - suited for large-scale systems

Invalidation vs. Update Protocols

- Invalidation-based
  - only write misses hit the bus (suited for write-back caches)
  - subsequent writes to same cache-line are write-hits
  - Good for multiple writes to the same cache line by the same CPU

- Update-based
  - all sharers of the cache line continue to hit in the cache after a write by one cache
  - Good for large-scale producer-consumer code
  - Otherwise lots of useless updates (wastes bandwidth)

- Hybrid forms are possible

MESI Cache Coherency Protocol

Developed at University of Illinois (1984)

- Modified (M)
  - Cache has only copy and copy is modified
  - Memory is not up-to-date
- Exclusive (E)
  - Cache has only copy and copy is unmodified (clean)
  - Memory is up-to-date
- Shared (S)
  - Copies may exist in other caches and all are unmodified
  - Memory is up-to-date
- Invalid (I)
  - Not in Cache

MESI Cache Coherency Protocol (Processor Transitions)

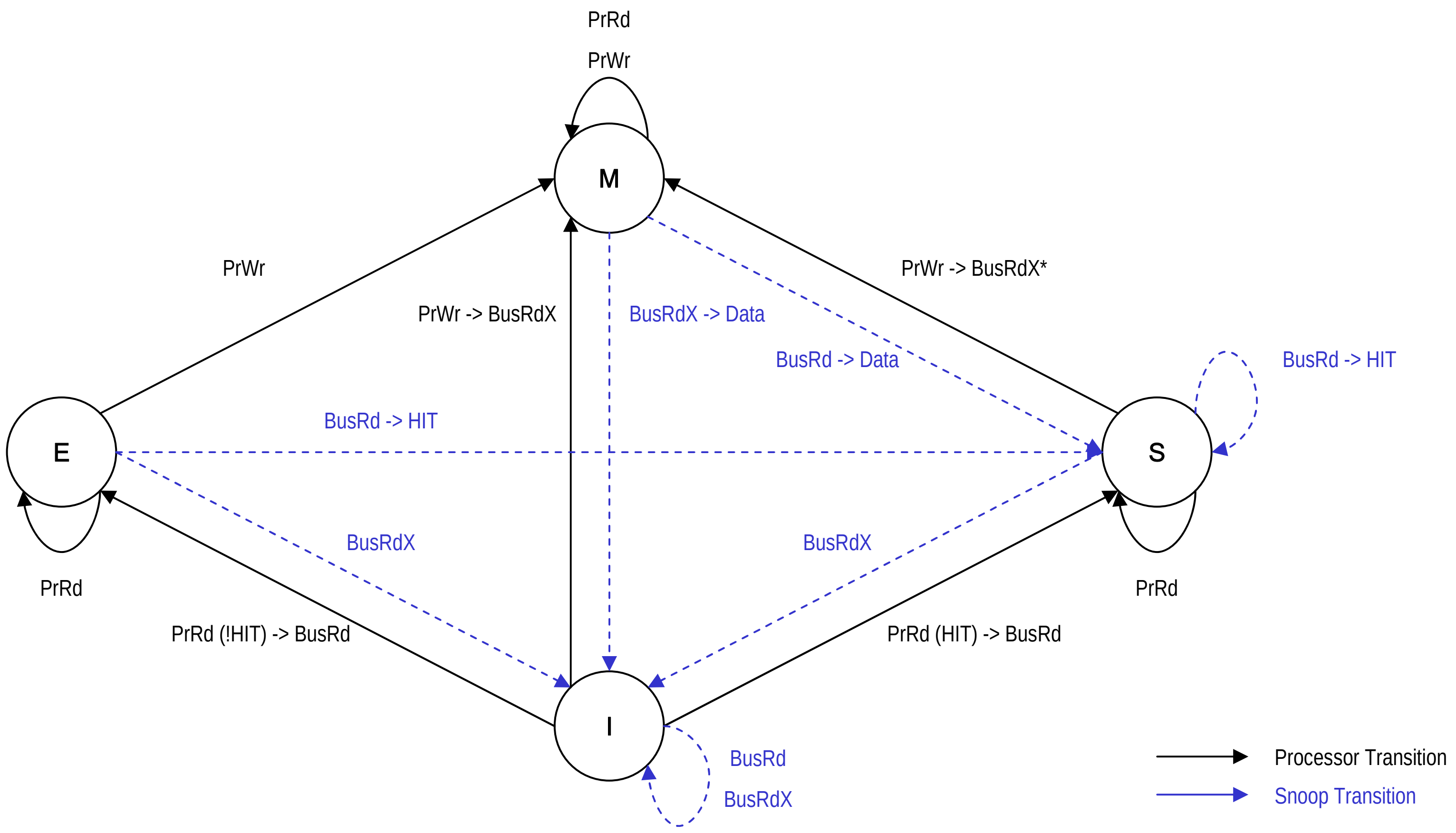
- State is I, CPU reads (PrRd)
  - Generate bus read request (BusRd), other caches signal sharing
  - If cache line in another cache go to S, otherwise transition to E
- State is S, E or M, CPU reads (PrRd)
  - No bus transaction, cache line already cached
- State is I, CPU writes (PrWr)
  - Generate bus read request for exclusive ownership (BusRdX)
  - transition to M
- State is S, CPU writes (PrWr)
  - Cache line already cached, but need to upgrade it for exclusive ownership (BusRdX*), transition to M
- State is E or M, CPU writes (PrWr)
  - No bus transaction, cache line already exclusively cached
  - transition to M

MESI Cache Coherency Protocol (Snoop Transitions)

- Receiving a read snoop (BusRd) for a cache line
  - If cache line is in cache (E or S), tell the requesting cache that the line is going to be shared (HIT signal) and transition to S
  - If cache line is modified in cache (M), supply the cache line to the requesting cache, inhibit memory from sending the cache line (HITM signal) and transition to S

- Receiving a read for exclusive ownership snoop (BusRdX) for a cache line
  - If cache line is modified in cache (M), write the cache line back to memory (Data), discard it and transition to I
  - If cache line is unmodified (E or S), discard it and transition to I

MESI Cache Coherency Protocol



MOESI Cache Coherency Protocol

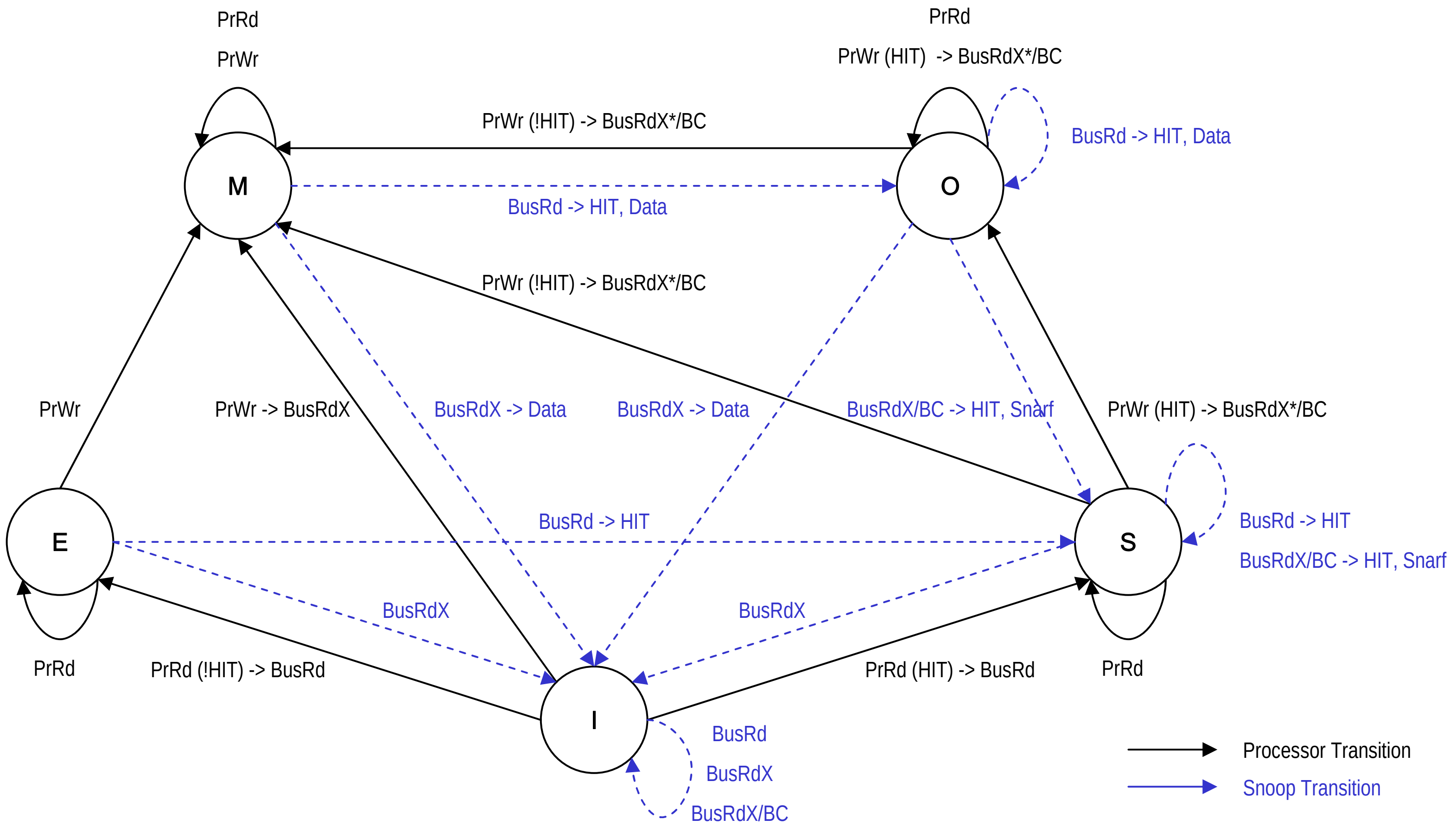
Developed for IEEE Futurebus (1986)

- Modified (M)
  - Exclusive/Modified, Memory not up-to-date, Cache supplies copy
- Owner (O)
  - Shared/Modified, Memory not up-to-date, Cache supplies copy
  - Copies may exist in other caches
- Exclusive (E)
  - Exclusive/Unmodified, Memory is up-to-date
- Shared (S)
  - Shared/Unmodified, Memory is up-to-date
  - Copies may exist in other caches
- Invalid (I)

MOESI Cache Coherency Protocol (Transitions)

- Similar to MESI, with some extensions
- BusRdX/BC is read for exclusive ownership with subsequent broadcast of new value to all other caches
- BusRdX*/BC is upgrade for exclusive ownership with subsequent broadcast of new value to all other caches
  - implemented as bus transaction with read length 0
- Cache-to-Cache transfers of modified cache lines
  - Cache in M or O state always supplies cache line (Data) instead of memory
  - Other caches read (Snarf) data off the bus and update themselves

MOESI Cache Coherency Protocol



Coherency in Multi-Level Caches

- Bus only connected to last-level cache (e.g., L2)
- Problem:
  - Snoop requests are relevant to inner-level caches (e.g, L1)
  - Modifications made in L1 may not be visible to L2 (and the bus)
- L1 intervention:
  - on BusRd check if cache line is M in L1 (may be E or S in L2)
  - on BusRdX send invalidation to L1
- Some interventions not needed when L1 is write-through
  - but causes more write traffic to L2