TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Distributed Operating Systems
Memory Consistency & Cache Coherence

Michael Roitzsch

(slides Julian Stecklina, Marcus Völp)

Concurrent programs

global variables:

```
int i;
int k;
```

```
i = 1;
if (i > 1) k = 3;
```

||

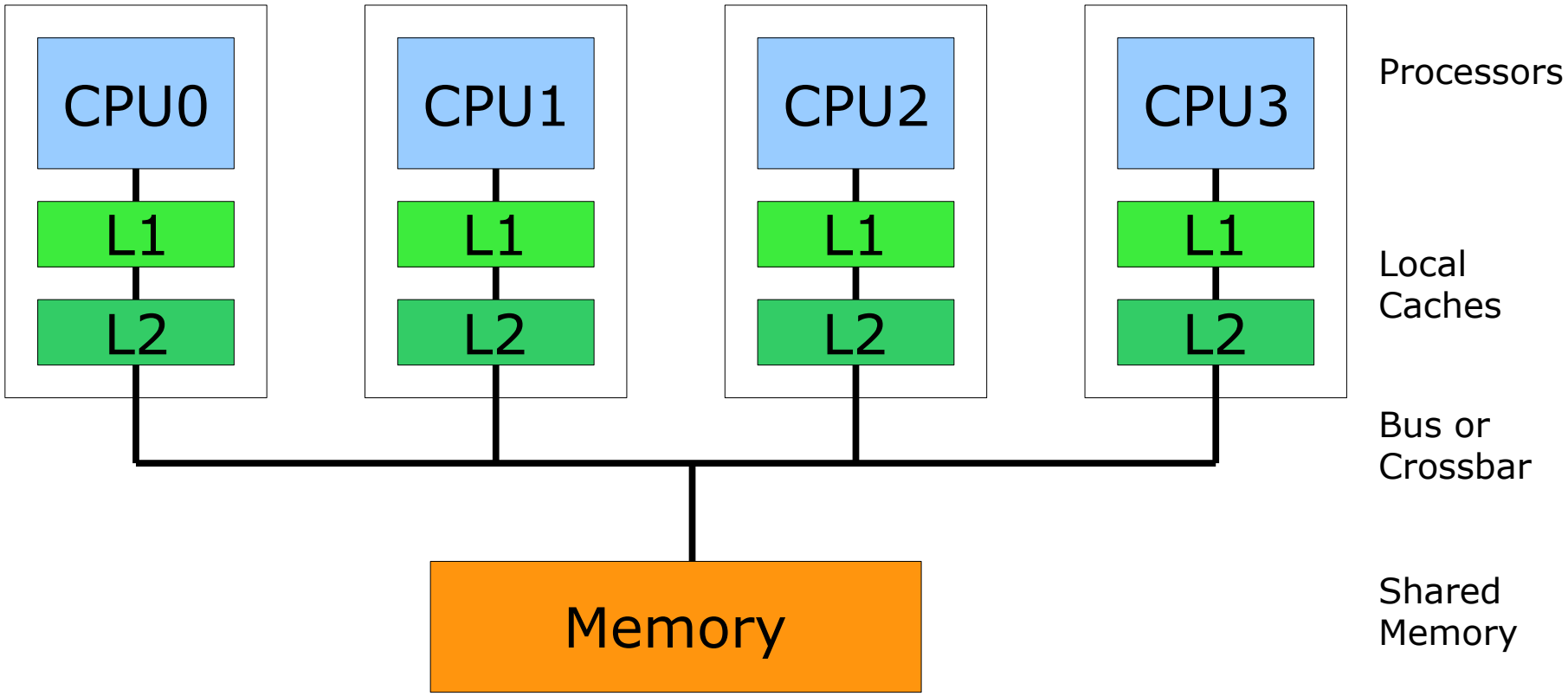
```
i = i + 1;
if (k == 0) k = 4;
```

```
mov $1, [%i]
cmp [%i], $1
jgt    end
mov $3, [%k]
end:
```

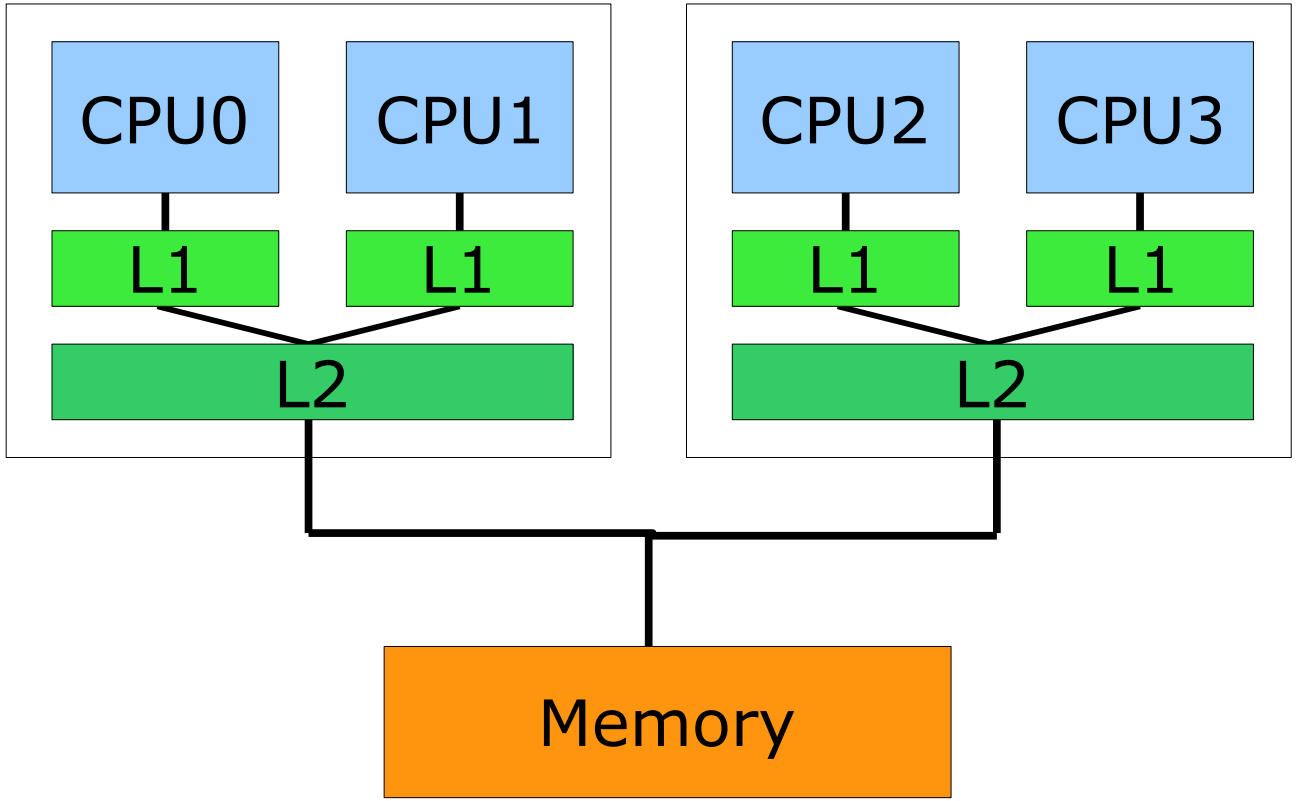
||

```
inc   [%i]
cmp [%k], $0
jne   end
mov $4, [%k]
end:
```

Symmetric Multiprocessor (SMP)



Chip Multiprocessor (CMP), Multicore



Processors

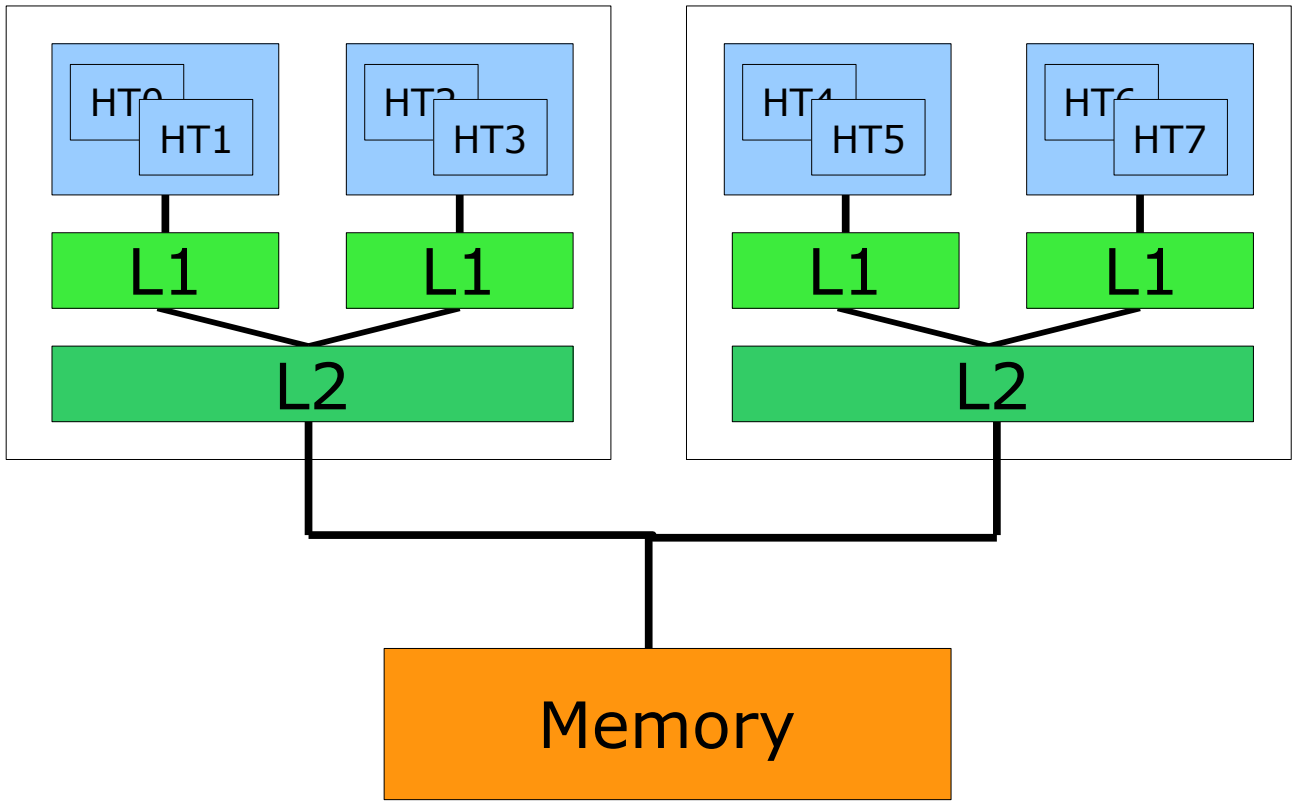
Local Cache

Shared LL-Cache

Bus or Crossbar

Shared Memory

Symmetric Multithreading (SMT), Hyperthreading



Processors

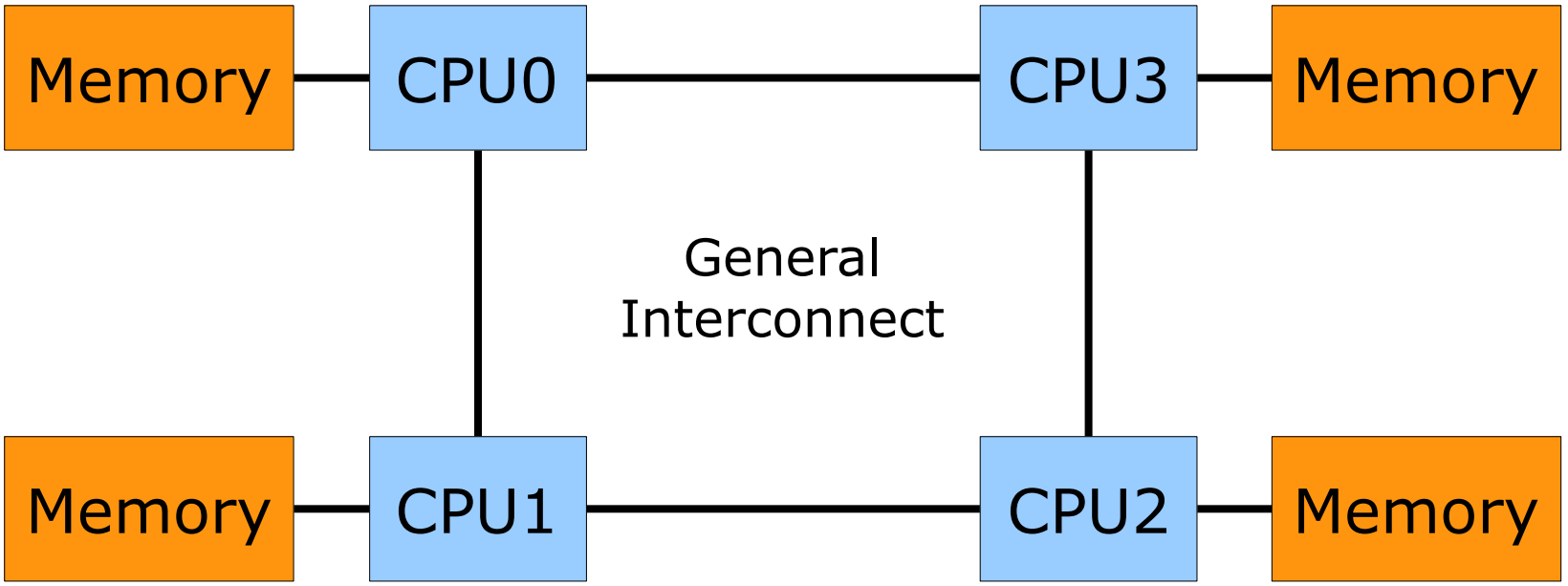
Local Cache

Shared LL-Cache

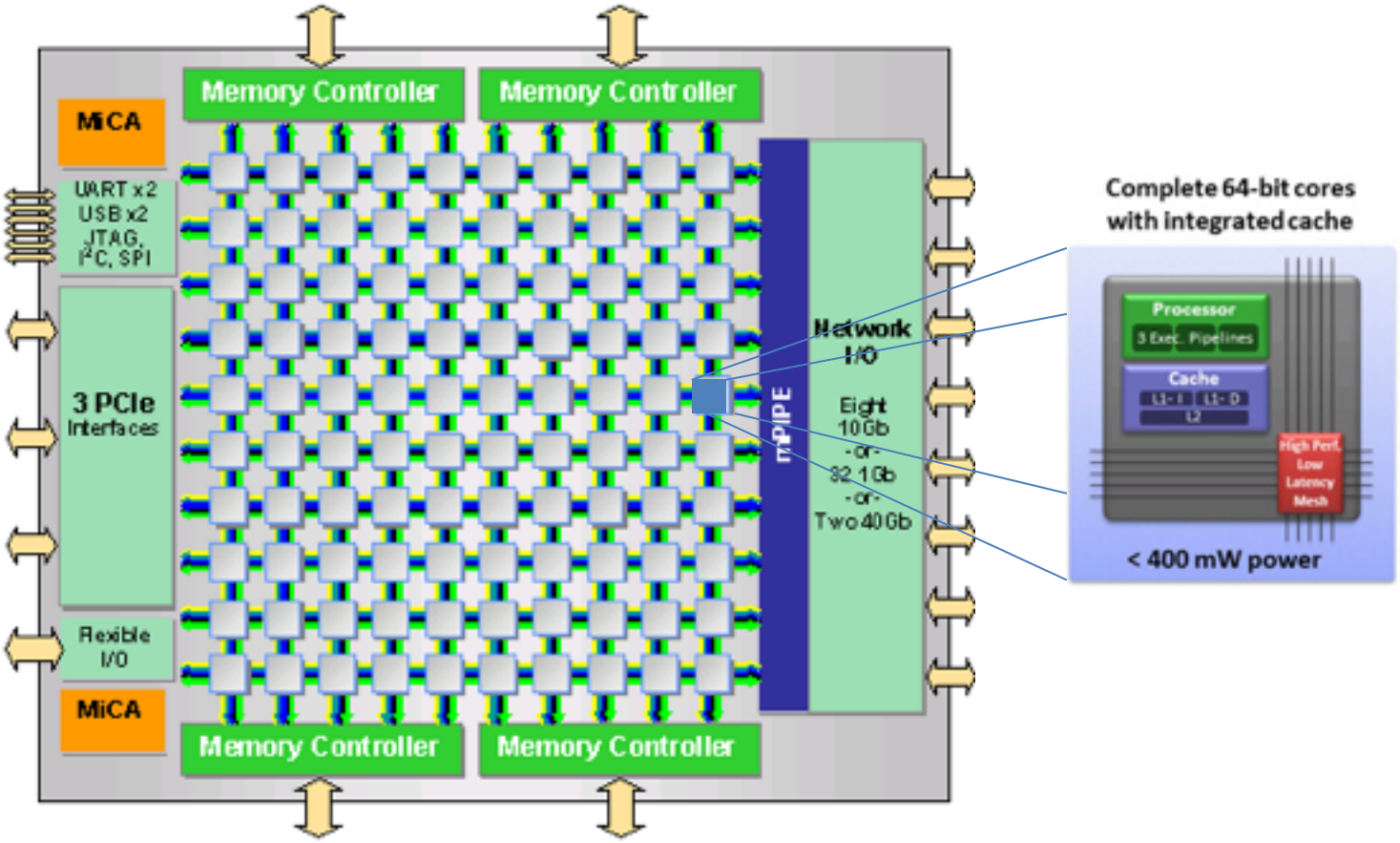
Bus or Crossbar

Shared Memory

Non-Uniform Memory Access (NUMA)



NUMA Example: Tilera Gx

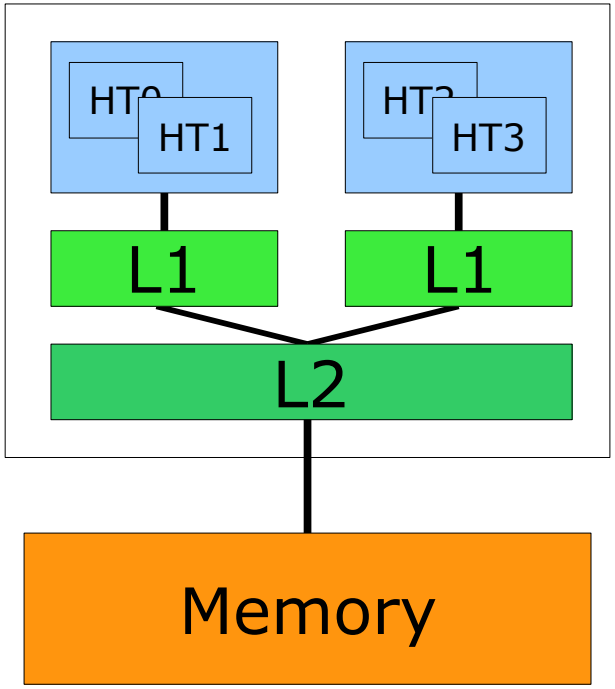


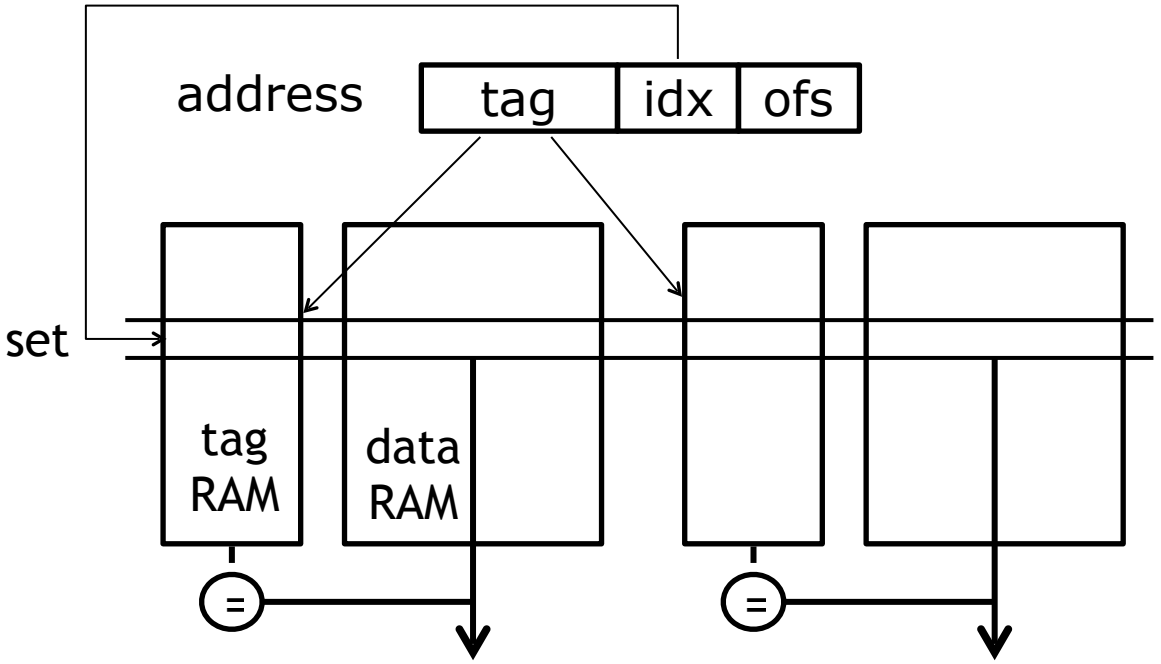
Source: http://tilera.com

Summary: Memory Organization

- Multiple processors share memory
- Memory access paths through one or more controllers
  - UMA (Uniform Memory Access)
  - NUMA (Non-Uniform Memory Access)
- Caches / store buffers hold memory content near accessing CPUs.

Cache Coherency





Cache Coherency

- Caches lead to multiple copies for the content of a single memory location
- Cache Coherency keeps copies “consistent”
  - locate all copies
  - invalidate/update content
- **Write Propagation**
  writes must eventually become visible to all processors.
- **Write Serialization**
  every processor should see writes to the **same** location in the same order.

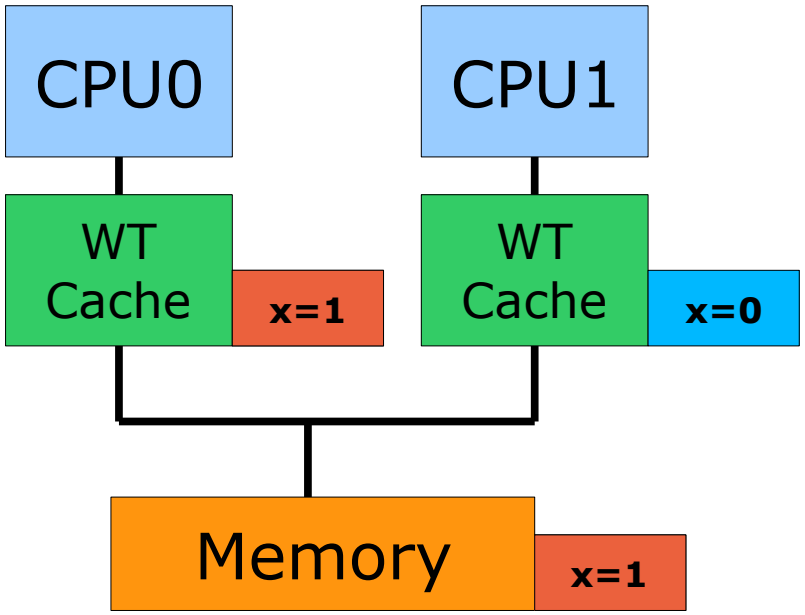
Alternative Definition: SWMR

Single-Writer, Multiple-Reader Invariant
For any memory location A, at any given time,
either only a single core may write (or read-modify-write) the content of A
or any number of cores may read the content of A.

Data-Value Invariant
The value of a memory location at the start of an operation is the same as the value at the end of its ***last*** write (read-modify-write) operation.

[based on Sorin et al., 2011]

Attempt 1: write through all caches



CPU0: read x
x=0 stored in cache

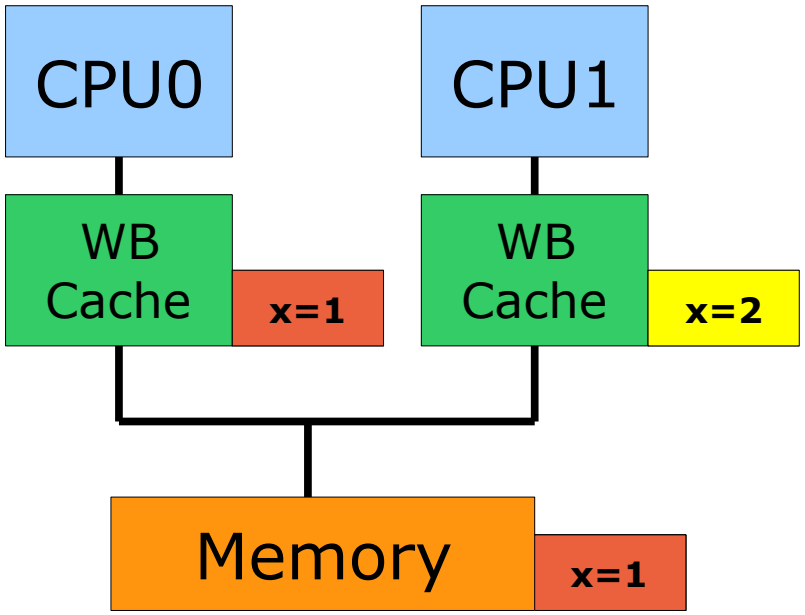
CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache
x=1 stored in memory

CPU1: read x
x=0 retrieved from cache

Write not visible to CPU1!

Attempt 2: write back



Later store x=2 lost!

CPU0: read x
x=0 stored in cache

CPU1: read x
x=0 stored in cache

CPU0: write x=1
x=1 stored in cache

CPU1: write x=2
x=2 stored in cache

CPU1: writeback
x=2 stored in memory

CPU0: writeback
x=1 stored in memory

Coherency Problems & Solutions

Both examples violate SWMR!

Problem 1

CPU1 used stale value that had already been modified by CPU0.

- Solution: Invalidate copies before write proceeds!

Problem 2

Incorrect writeback order of modified cache lines.

- Solution: Disallow more than one modified copy!

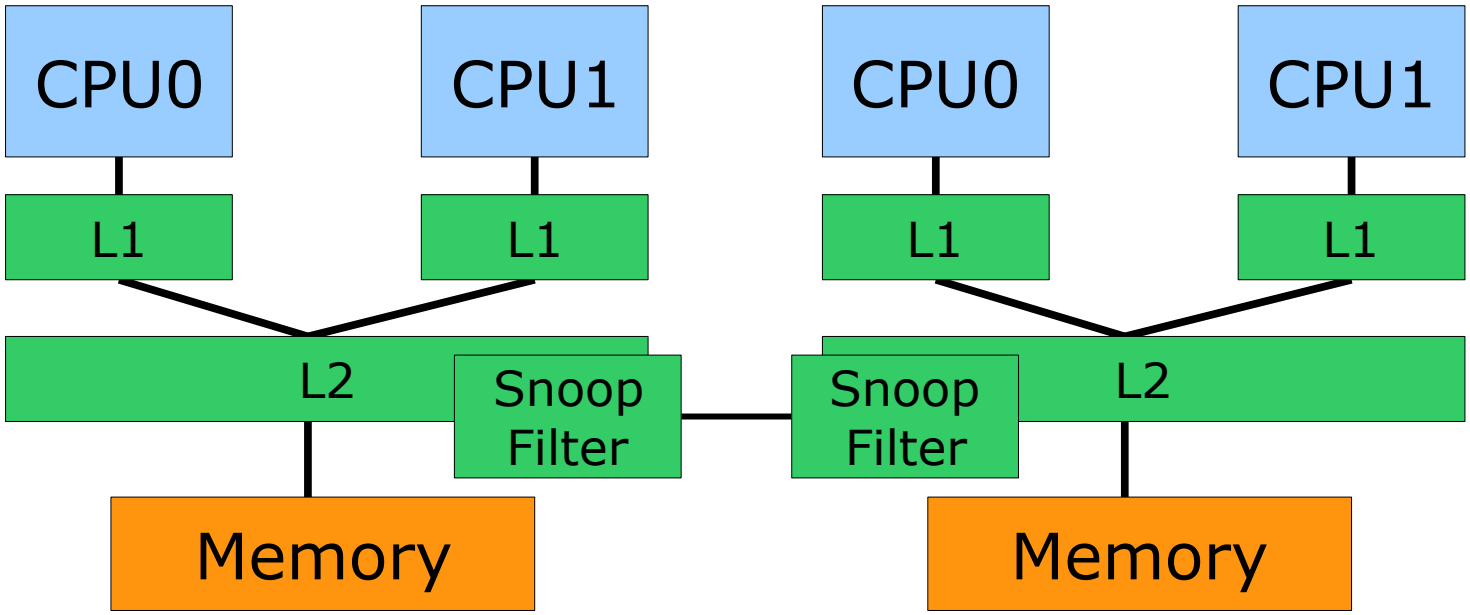
Coherency Protocol Design Space

- **Snooping-based**
  - All coherency related traffic broadcasted to all CPUs
  - Each processor snoops and acts accordingly:
    - Invalidate lines written by other CPUs
    - Signal sharing for lines currently in cache
  - Straightforward for bus-based systems
  - Suited for small-scale systems

- **Directory-based**
  - Uses central directory to track cache line owner
  - Update copies in other caches
    - Can update all CPUs at once
      (less traffic for alternating reads and writes)
    - Multiple writes need multiple updates
      (more traffic for subsequent writes)
  - Suited for large-scale systems

Coherency Protocol Design Space

- **Snooping-based vs. Directory-based**



Invalidation vs. Update Protocols

- **Invalidation-based**
  - Only write misses hit bus (suited for WB caches)
  - Subsequent writes are write hits
  - Good for multiple writes to same cache line by same CPU
- **Update-based**
  - All shares of a cache line continue to hit in the cache after a write by one CPU
  - Otherwise lots of useless updates (wastes bandwidth) → Rarely used!

- Hybrid forms are possible!

A Basic Coherency Protocol: MSI

- Modified (M)
  - No copies on other caches; local copy modifed
  - Memory is stale
- Shared (S)
  - Unmodified copies in one or more caches
  - Memory is up-to-date
- Invalid (I)
  - Not in cache

- States tracked from the view of the cache controller. Sees events from:
  - Local processor → processor transactions
  - Other processors → snoop transactions

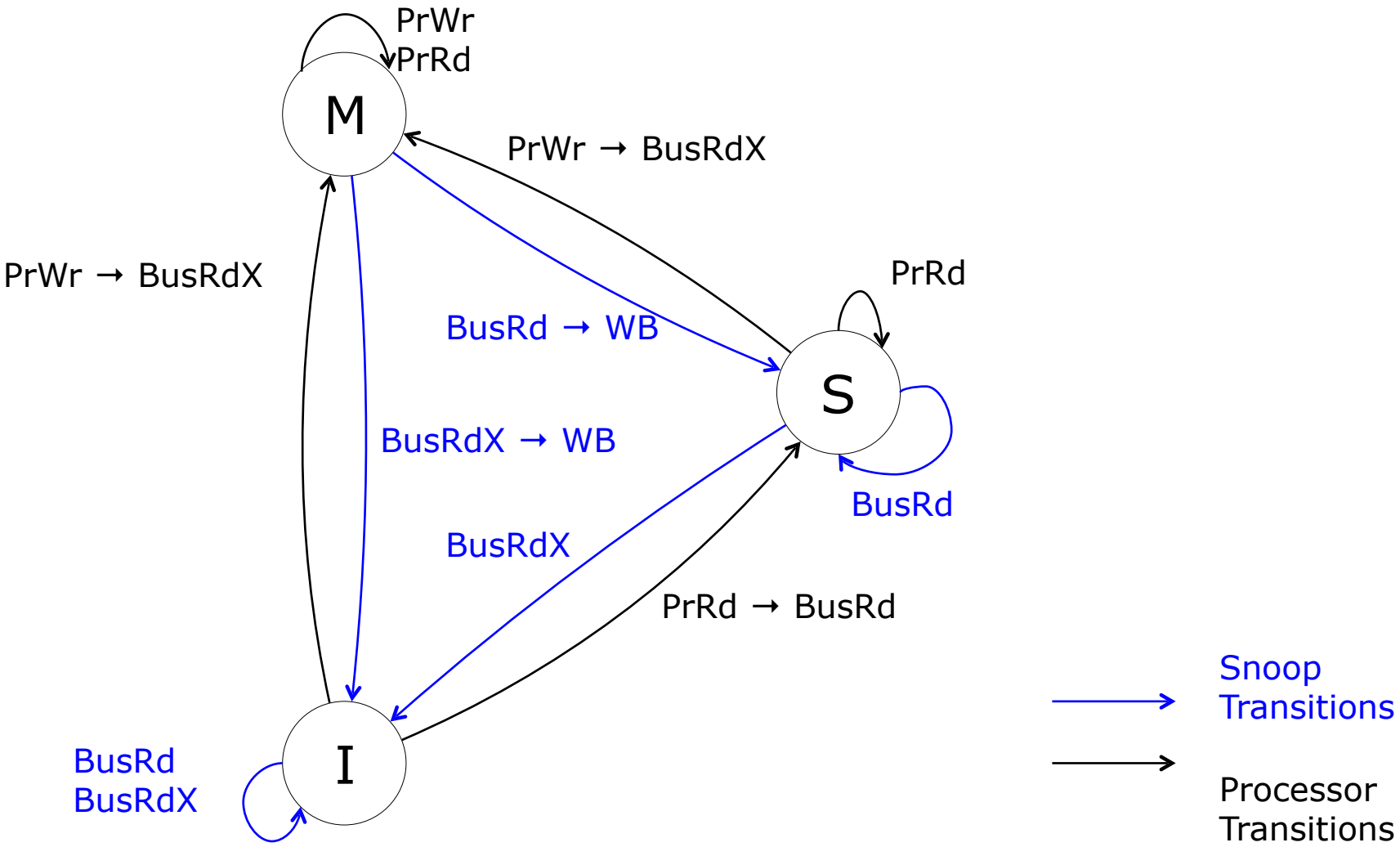
MSI: Processor Transitions

- State is I, CPU reads (PrRd)
  - Generate bus read request (BusRd)
  - Go to S
- State is S or M, CPU reads (PrRd)
  - No transition
- State is S, CPU writes (PrWr)
  - Upgrade cache line for exclusive ownership (BusRdX)
  - Go to M
- State is M, CPU writes (PrWr)
  - No transition

MSI: Snoop Transitions

- Receiving a read snoop (BusRd) for a cache line
  - If M, write cache line back to memory (WB), transition to S
  - If S, no transition
- Receiving a exclusive ownership snoop (BusRdX)
  - If M, write cache line back to memory (WB), discard it, transition to I
  - If S, discard cache line, transition to I

MSI State Transitions



Problems in MSI

A common usecase is to:

- read variable A: S
- Modify A: BusRdX sent, S → M

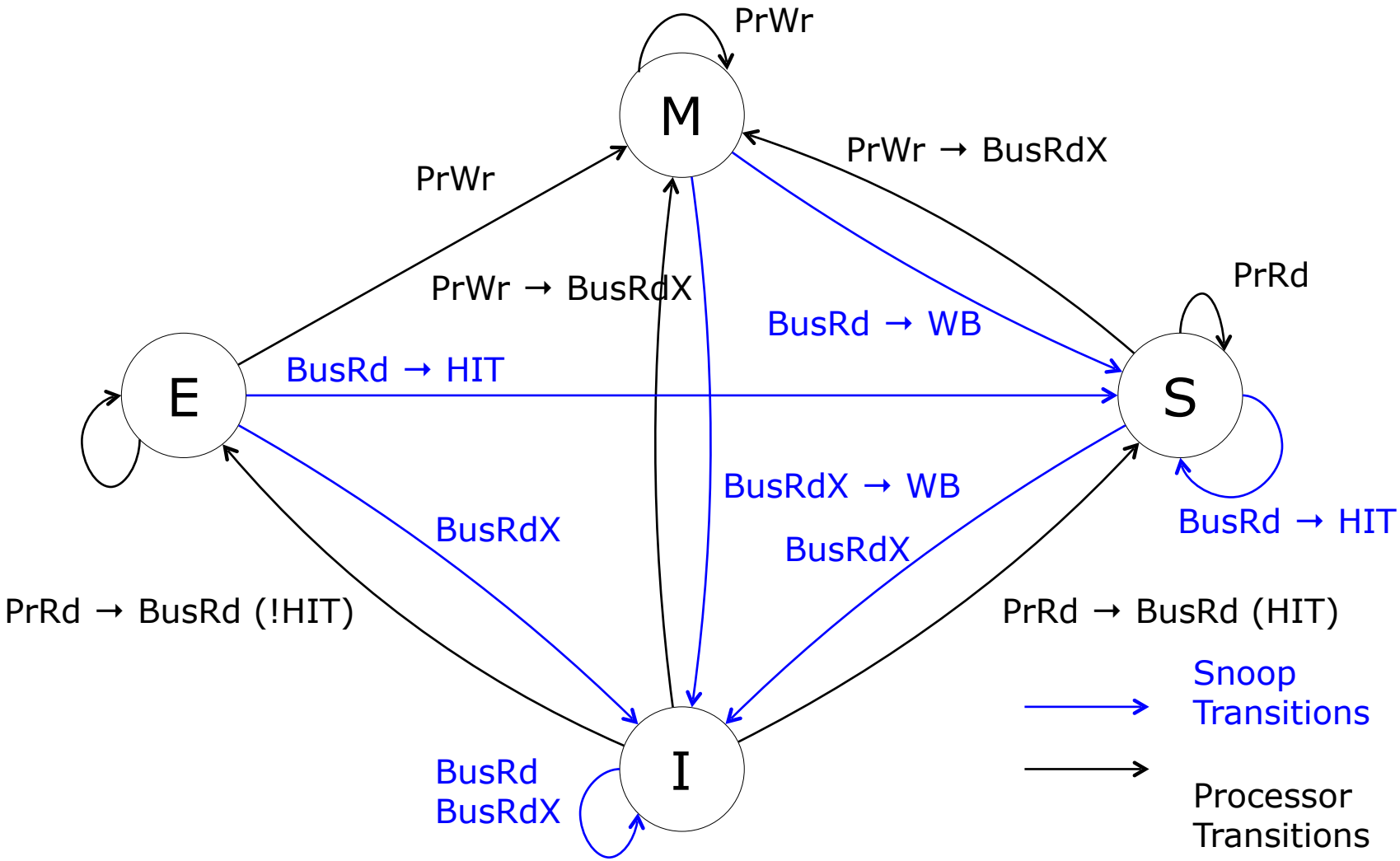
Invalidation message pointless, if no other cache holds A.

Solved by adding Exclusive (E) state:

- No copies exist in other caches
- Memory is up-to-date

Variants of MESI are used by most popular microprocessors.

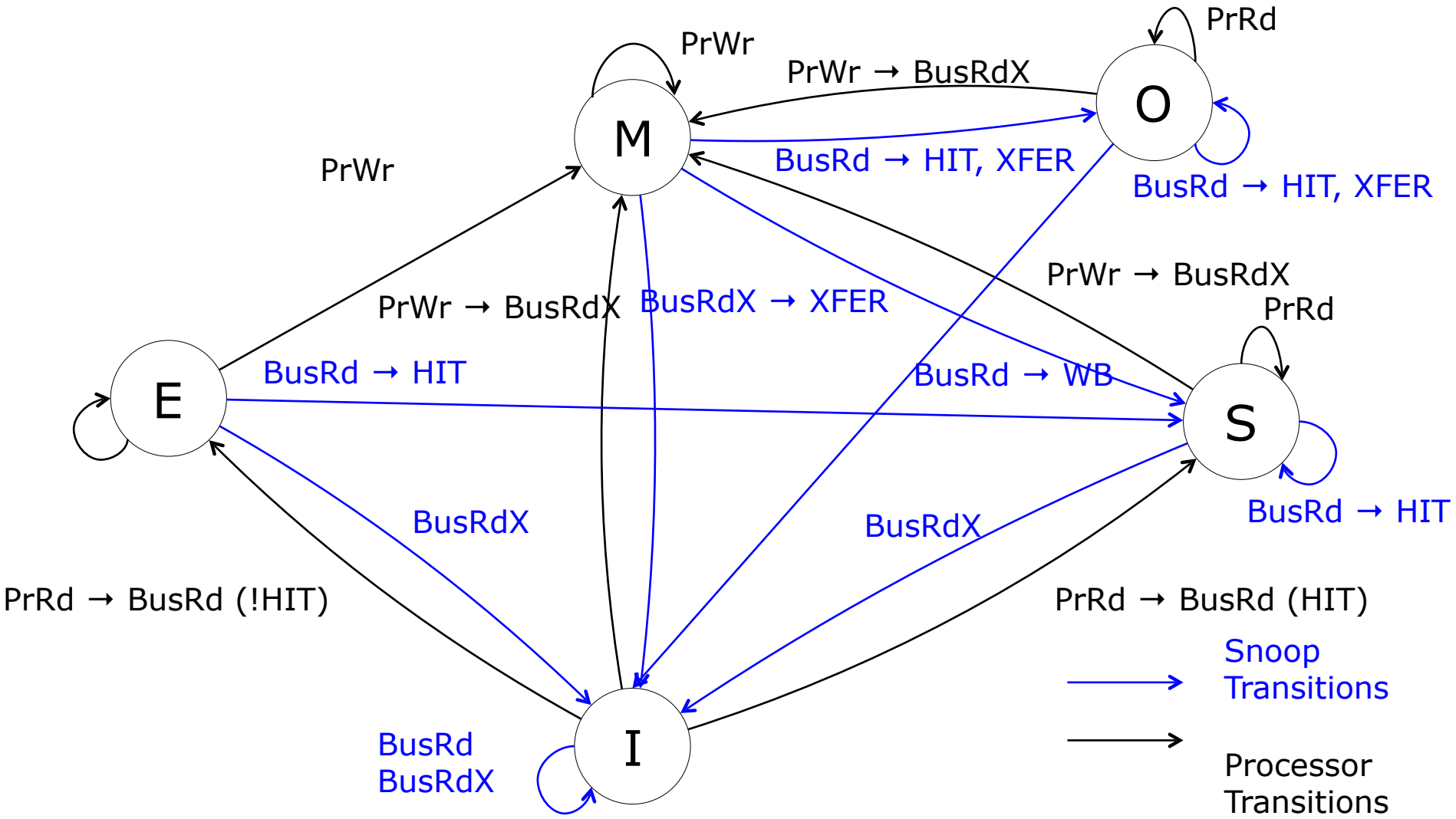
MESI State Transitions



MOESI: Adding Owned to MESI

- Similar to MESI, with some extensions
- Cache-to-Cache transfers of modified cache lines
  - Modified cache lines not written back to memory, but supplied by to other CPUs on BusRd
  - CPU that had initial modified copy becomes “owner”
- Avoids writeback to memory when another CPU accesses cache line
  - Beneficial when cache-to-cache latency/bandwidth is better than cache-to-memory latency/bandwidth
- Used by AMD Opteron

MOESI State Transitions



Coherency in Multi-Level Caches

- Bus only connected to last-level cache (e.g. L2)
  - Snoop requests are relevant to inner-level caches (e.g. L1)
  - Modifications in L1 may not be visible to L2 (and the bus)
- Idea: L2 forwards filtered transactions for L1:
  - On BusRd check if line is M/O in L1 (may be S or E in L2)
  - On BusRdX, send invalidate to L1
- Only easy for inclusive caches!

- **Inclusion property**
  Outer cache contains a superset of the content of its inner caches.

Concurrent programs

global variables:

```
int i;
int k;
```

```
i = 1;
if (i > 1) k = 3;
```

||

```
i = i + 1;
if (k == 0) k = 4;
```

```
mov $1, [%i]
cmp [%i], $1
jgt    end
mov $3, [%k]
end:
```

||

```
lock; inc   [%i]
      cmp [%k], $0
      jne   end
      mov $4, [%k]
      end:
```

Memory Consistency Models

Memory Consistency Model

defines correct shared memory behavior in terms of loads and stores in terms of how operations to different memory locations may become visible with respect to each other.

Different memory consistency models exist

- Complex models can expose more performance
- Some platforms support multiple models (SPARC)

Terminology

- **Program Order** (of a processor's operations)

  Per-processor order of memory accesses determined by the program (software)

- **Visibility Order** (of all operations)

  Order of memory accesses observed by one or more processors.

Sequential Consistency (SC)

"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. A multiprocessor satisfying this condition will be called **sequentially consistent**." [Lamport 1979]

- Program Order Requirement
  - Each CPU issues memory operations in program order.
- Atomicity Requirement
  - Memory services operations one at a time
  - Memory operations appear to execute atomically wrt other memory operations
- Implemented by MIPS R10k

Examples for Sequential Consistency

CPU0	**CPU1**	
[A] = 1; (a1)	u = [B]; (a2)	[A] [B] Memory
[B] = 1; (b1)	v = [A]; (b2)	u, v Registers

(u,v) = (1,1)

- Sequentially consistent: a1, b1, a2, b2

(u,v) = (1,0)

- Not sequentially consistent: b1, a2, b2, a1
- Violates program order for CPU0 (or 1)
- No visibility order possible that is seq. consistent!

Examples for Sequential Consistency

CPU0	**CPU1**	
[A] = 1; (a1)	[B] = 1; (a2)	[A] [B] Memory
u = [B]; (b1)	v = [A]; (b2)	u, v Registers

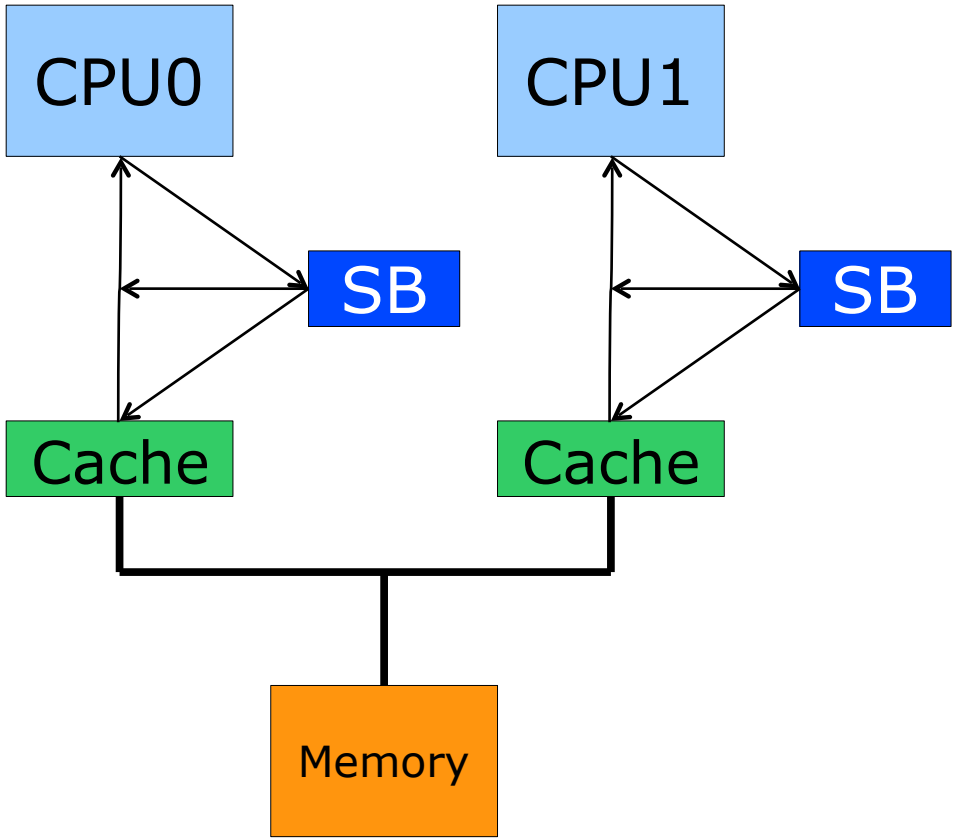
(u,v) = (1,1)

- Sequentially consistent: a1, a2, b1, b2

(u,v) = (0,0)

- Not sequentially consistent: b1, b2, a1, a2
- Violates program order for CPU0/1
- No visibility order possible that is seq. Consistent!

Store Buffer



SB optimizes writes to memory and/or caches to optimize interconnect accesses.

CPU can continue before write is completed.

Store forwarding allows reads from local CPU to see pending writes in the SB.

SB invisible to remote CPUs.

FIFO vs. non-FIFO. Writes can be combined, may reorder writes on some architectures.

SC vs. weaker consistency models

In-order memory operations in SC:

- Read→Read
- Read→Write
- Write→Read
- Write→Write

Describes which program order relations hold in the visibility order of memory operations.

Weaker models relax some or all of these orderings.

Relaxing Write→Read or Write→Write

Relaxing Write→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 cache line ownership
  - Reads can bypass writes in the buffer
  - Hides write latency

Relaxing Write→Write (later writes can bypass earlier writes)

- Write followed by a write can execute out-of-order
- Typical hardware usage: Coalescing store buffer

IBM 370 (z Series)

- In-order memory operations:
  - Read→Read
  - Read→Write
  - Write→Write
- Out-of-order memory operations:
  - Write-to-Read (later reads can bypass earlier writes)
    - Unless both to same location
    - Breaks Dekker's algorithm for mutual exclusion
  - Write-to-Read to same location must execute in-order
    - No forwarding from the store buffer

Dekker's Algorithm on z Series

```
bool flag0 = false; //
Intention
bool flag1 = false; // to enter
int turn = 0; // Who's next?
```

CPU0

```
P: flag0 = true;
while (flag1) {
   If (turn == 1) {
    flag0 = false;
    goto P;
   }
 }
// Critical section
flag0 = false;
turn = 1;
```

Buffered

CPU1

```
P: flag1 = true;
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 reads to the same location
      - Writes become visible to writing processor first
    - Store buffer is FIFO
    - Breaks Peterson's algorithm for mutual exclusion

Peterson's Algorithm on TSO

```
bool flag0 = false; //
Intention
bool flag1 = false; // to enter
int turn = 0; // Who's next?
```

CPU0

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

Buffered

CPU1

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

Loading *turn* orders accesses on zSeries, but not on TSO!

TSO vs. SC and z Series

CPU0	**CPU1**
[A] = 1; (a1)	[B] = 1; (a2)
u = [A]; (b1)	v = [B]; (b2)
w = [B]; (c1)	x = [A]; (c2)

- (u,v,w,x) = (1,1,0,0)
  - Not possible with SC and z Series
  - Possible with TSO
    - b1, b2, c1, c2, a1, a2
    - b1 reads [A] from write buffer
    - b2 reads [B] from write buffer

Processor Consistency (PC)

- Similar to Total Store Order (TSO)
- Additionally supports multiple cached memory copies
  - Relaxed atomicity for write operations
    - Each write broken into suboperations to update cached copies of other CPUs
  - Non-unique write order: **per-CPU visibility order**

- Additional coherency requirement
  - All write suboperations to the same location complete in the same order across all memory copies (or in other words: each processor sees writes to the same location in the same order)

PC vs. SC, z Series, TSO

CPU0	**CPU1**	**CPU2**
[A] = 1; (a1)	u = [A]; (a2) [B] = 1; (b2)	v = [B]; (a3) w = [A]; (b3)

- (u,v,w) = (1,1,0)
  - Not possible with SC, z Series, TSO
  - Possible with Processor Consistency (PC)
    - CPU0 sets [A], sends update to other CPUs
    - CPU1 gets update, sets [B], sends update
    - CPU2 sees update from CPU1, but hasn't seen update from CPU0 yet
  - Single memory bus enforces single visibility order
  - Multiple visibility orders with different topologies

Causality

CPU0	**CPU1**	**CPU2**
[A] = 1;	while ([A] == 0);	while ([B] == 0);
	[B] = 1;	print [A];

Write Atomicity

All cores see writes at the same time (and the same order).

Relaxing write atomicity

- CPU0 writes [A]; sends update to CPU1/2
- CPU1 receives; writes [B]; sends update to CPU2
- CPU2 receives update from CPU1, prints [A] = 0
- CPU2 receives update from CPU0

Not sequentially consistent!

SPARC V8 Partial Store Order (PSO)

- In-order memory operations:
  - Read→Read
  - Read→Write
- Out-of-order memory operations:
  - Write→Read (later reads can bypass earlier writes)
    - Forwarding of pending writes to successive reads to the same location
  - Write→Write (later writes can bypass earlier writes)
    - Unless both are to the same location
    - Breaks naive producer-consumer code
- Write atomicity is maintained → single visibility order

PSO vs. SC, z Series, TSO, PC

CPU0	**CPU1**	
[A] = 1;	(a1)	while ([Flag] == 0); (a2)
[B] = 1;	(b1)	u = [A]; (b2)
[Flag] = 1;	(c1)	v = [B]; (c2)

- (u,v) = (0,0) or (0,1) or (1,0)
  - Not possible with SC, z Series, TSO, PC
  - Possible with PSO
    - c1,a2,b2,c2,a1,b1
    - Store Barrier (STBAR) before c1 ensures sequentially consistent result (u,v) = (1,1)

Relaxing all Program Orders

- In addition to previous relaxations:
  - Read→Read (later reads can bypass earlier reads)
    - Read followed by read can execute out-of-order
  - Read→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 Model (RMO)
  - PowerPC
  - Itanium (IA-64)

Weak Ordering (WO)

- Conceptually similar to Processor Consistency
  - Including coherency requirement
- Classifies memory operations into
  - Data operations
  - Synchronization operations
- Reordering of operations 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, reading a pointer
  - Release
    - Write operation for granting permission to a shared resource
    - e.g., setting a synchronization flag

Flavors of Release Consistency

- 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** release followed by acquire

Enforcing Ordering: Synchronization Instructions

- IA32
  - lfence, sfence, mfence (load, store, 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)
- PowerPC
  - sync (similar to Alpha mb, except r-r), lwsync
  - eieio (enforce in-order execution of I/O)

Dependent Load Reordering (Alpha)

A = 1, B = 0, P = &A

CPU0		**CPU1**	
[B] = 1;	(a1)	u = [P]	(a2)
Store barrier		v = [u];	(b2)
[P] = &B;	(b1)		

Load depends previous load for address generation. Alpha may reorder loads due to speculation. Allows:

(u,v) = (&B, 0)

- Even with barrier between a1,b1!
- Visibility order: a1,b1,b2,a2

Most (all?) processors except Alpha disallow dependent load/store reordering.

Compiler Optimizations

Compilers reorder memory accesses for performance.
Effects are equivalent to reordering by hardware.

```
Flag0 = true;          ld r1 ← flag1
while (flag1) {                st flag0 ← true
 …                     loop: cmp r1,0
 …                     ...
}                      ld r1 ← flag1
```

Is this a legal optimization?

Single threaded: Yes Multithreaded: NO!

Can't perceive difference

How do we portably program this?!

Standardized memory models for HLL:

- C / C++ 2011
- Java

Basic model: Sequentially Consistency for data-race free programs (SC-DRF)

A DRF program will execute sequentially consistent.

Data Race (informal)

Multiple threads access a memory location without synchronization, one of them is a writer.

DRF Example

a = b = 0;

Thread 1	**Thread 2**
mtx_lock(l);	
a = 1;	x = a;
b = 1;	y = b;
mtx_unlock(l);	

Not DRF:

- a,b accessed without synchronization
- (x,y) = (0,0) (1,0) (0,1) (1,1) all legal!
- Need to add synchronization to Thread 2

With synchronization yields either (0,0) or (1,1):

- DRF, sequentially consistent!

Enforcing Memory Ordering in C++

- Mutexes may cause scalability issues
- C++ 11 offers rich set of atomic memory operations (std::atomic)
  - Implements RC_{SC}:
    - Atomic reads acquire
    - Atomic stores release
  - Can use weaker ordering if desired
  - Compare-and-Swap
  - Add/Sub/And/Or/Xor/...
- Does the right thing on all platforms
  - Adds appropriate memory barriers
  - Uses locked instructions as necessary
  - May use locks on certain platforms!

References

- A Primer on Memory Consistency and Cache Coherence
  Sorin, Hill, Wood; 2011
- atomic<> Weapons: The C++ Memory Model and Modern Hardware (Video)
  Sutter; 2013
- Shared memory consistency models: a tutorial
  Adve, Gharachorloo; 1996
- IA Memory Model
  Richard Hudson; Google Tech Talk 2008
- Memory Ordering in Modern Microprocessors
  McKenney; Linux Journal 2005
- How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs
  Lamport, 1979
- PowerPC Storage Model