Computer Architecture

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

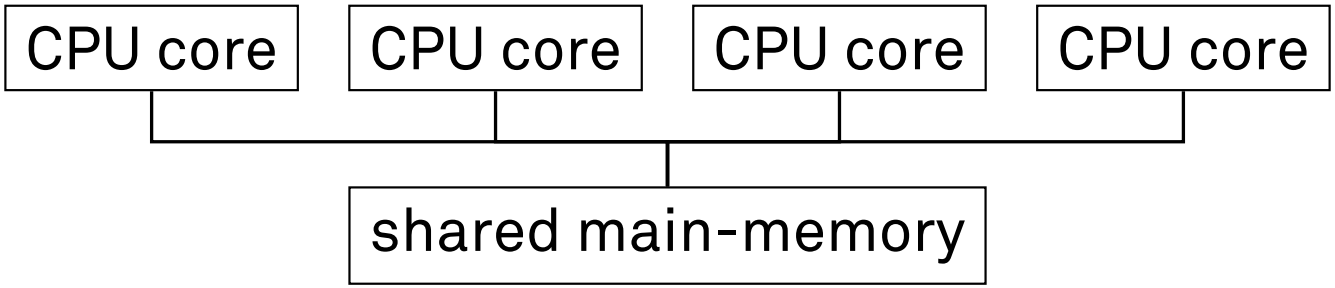
Summer 2016

Part III

Multi-Core Systems

Building a Shared-Memory Multiprocessor

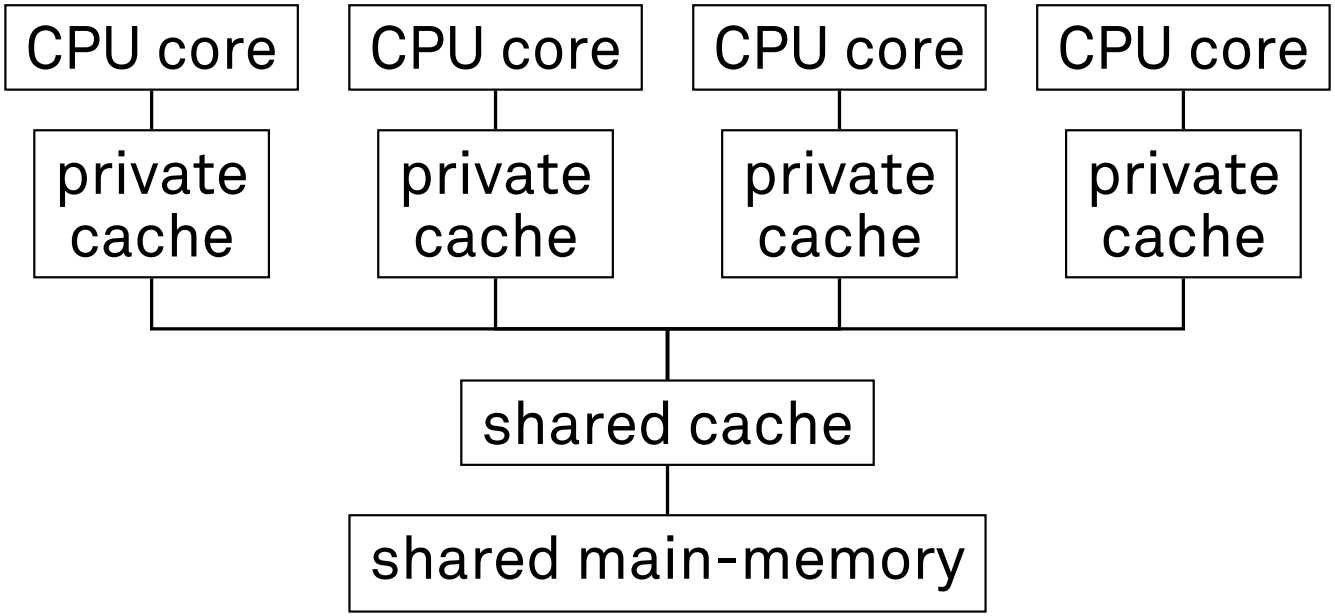
What the programmer likes to think of…



✎ **Scalability? Moore's Law?**

Centralized Shared-Memory Multiprocessor

Caches help mitigate the bandwidth bottleneck(s).



- A **shared bus** connects CPU cores and memory.
  - → the “shared bus” may or may not be shared physically.
- The Intel Core architecture, *e.g.*, implemented this design.

Centralized Shared-Memory Multiprocessor

The shared bus design with caches makes sense:

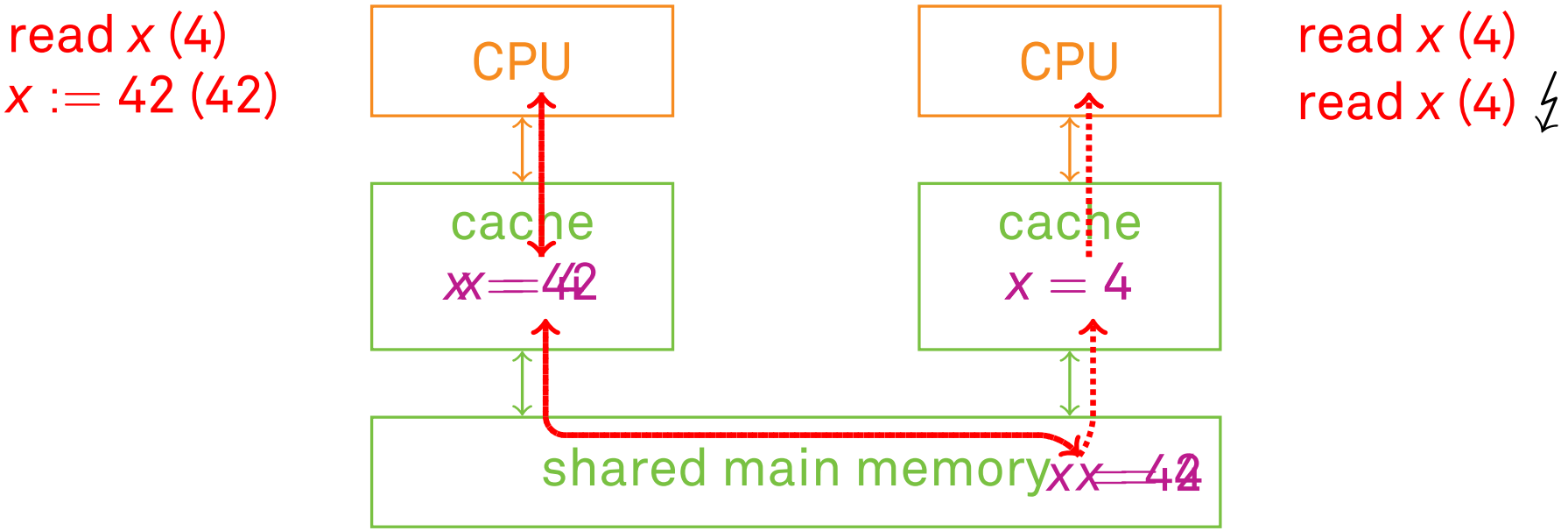
+ **symmetric design**; uniform access time for every memory item from every processor
+ **private data** gets **cached locally**
  → behavior identical to that of a uniprocessor
? **shared data** will be **replicated to private caches**
  → Okay for parallel **reads.**
  → But what about **writes** to the replicated data?
  → In fact, we'll want to use memory as a mechanism to communicate between processors.

The approach does have **limitations**, too:

– For **large core counts**, shared bus may still be a (bandwidth) bottleneck.

Caches and Shared Memory

Caching/replicating shared data can cause problems:



Challenges:

- Need **well-defined semantics** for such scenarios.
- Must **efficiently implement** that semantics.

Cache Coherence

The desired property (semantics) is **cache coherence.**

Most importantly:[5]

> *Writes to the **same location** are **serialized**; two writes to the same location (by any two processors) are seen in the same order by all processors.*

Note:

- We did not specify **which** order will be seen by the processors.
  → ✎ **Why?**

[5] We also demand that a read by processor P will return P's most recent write, provided that no other processor has written to the same location meanwhile. Also, every write must be visible by other processors after "some time."

Cache Coherence Protocol

Multiprocessor (or multicore) systems maintain coherence through a **cache coherence protocol**.

Idea:

- Know **which cache/memory** holds the **current value** of the item.
- Other replicas might be stale.

Two alternatives:

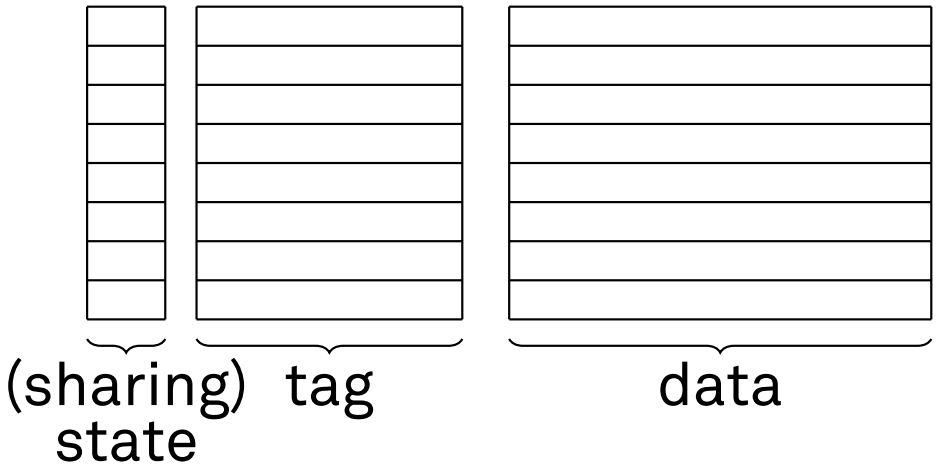
1. **Snooping-Based Coherence**
   → All processors communicate to agree on item states.
2. **Directory-Based Coherence**
   → A centralized **directory** holds information about state/whereabouts of data items.

Snooping-Based Cache Coherence

Rationale:

- All processors have access to a **shared bus.**
- Can "snoop" on the bus to track other processors' activities.

Use to track the **sharing state** of each cached item:



Meta data for each **cache block**:

- (sharing) state
- block identification (tag)

✎ **Ignoring Multiprocessors for a moment, which "state" information might make sense to keep?**

Strategy 1: Write Update Protocol

Idea:

- On **every write**, propagate the write to **every copy**.
  - → Use bus to **broadcast writes.**[6]

✎ **Pros/Cons of this strategy?**

[6] The protocol is thus also called *write broadcast* protocol.

Strategy 2: Write Invalidate Protocol

Idea:

- **Before writing** an item, **invalidate all other copies.**

Activity	Bus	Cache A	Cache B	Memory
				$x = 4$
A reads x	cache miss for x	$x = 4$		$x = 4$
B reads x	cache miss for x	$x = 4$	$x = 4$	$x = 4$
A reads x	– (cache hit)	$x = 4$	$x = 4$	$x = 4$
B writes x	invalidate x	~~$x = 4$~~	$x = 42$	$x = 4$[7]
A reads x	cache miss for x	$x = 42$	$x = 42$	$x = 42$

→ Caches will **re-fetch** invalidated items automatically.

- Since the bus is shared, other caches may answer “cache miss” messages ($\rightsquigarrow$ necessary for write-back caches).

[7] With write-through caches, memory will be updated immediately.

Write Invalidate—Realization

Realization:

- To **invalidate**, **broadcast** address on bus.
- All processors continuously **snoop on bus**:
  - *invalidate* message for address held in own cache
    - → Invalidate own copy
  - *miss* message for address held in own cache
    - → Reply with own copy (for write-back caches)
    - → Memory will see this and abort its own read

✎ **What if two processors try to write at the same time?**

Write Invalidate—Tracking Sharing States

Through snooping, can monitor all bus activities by all processors.

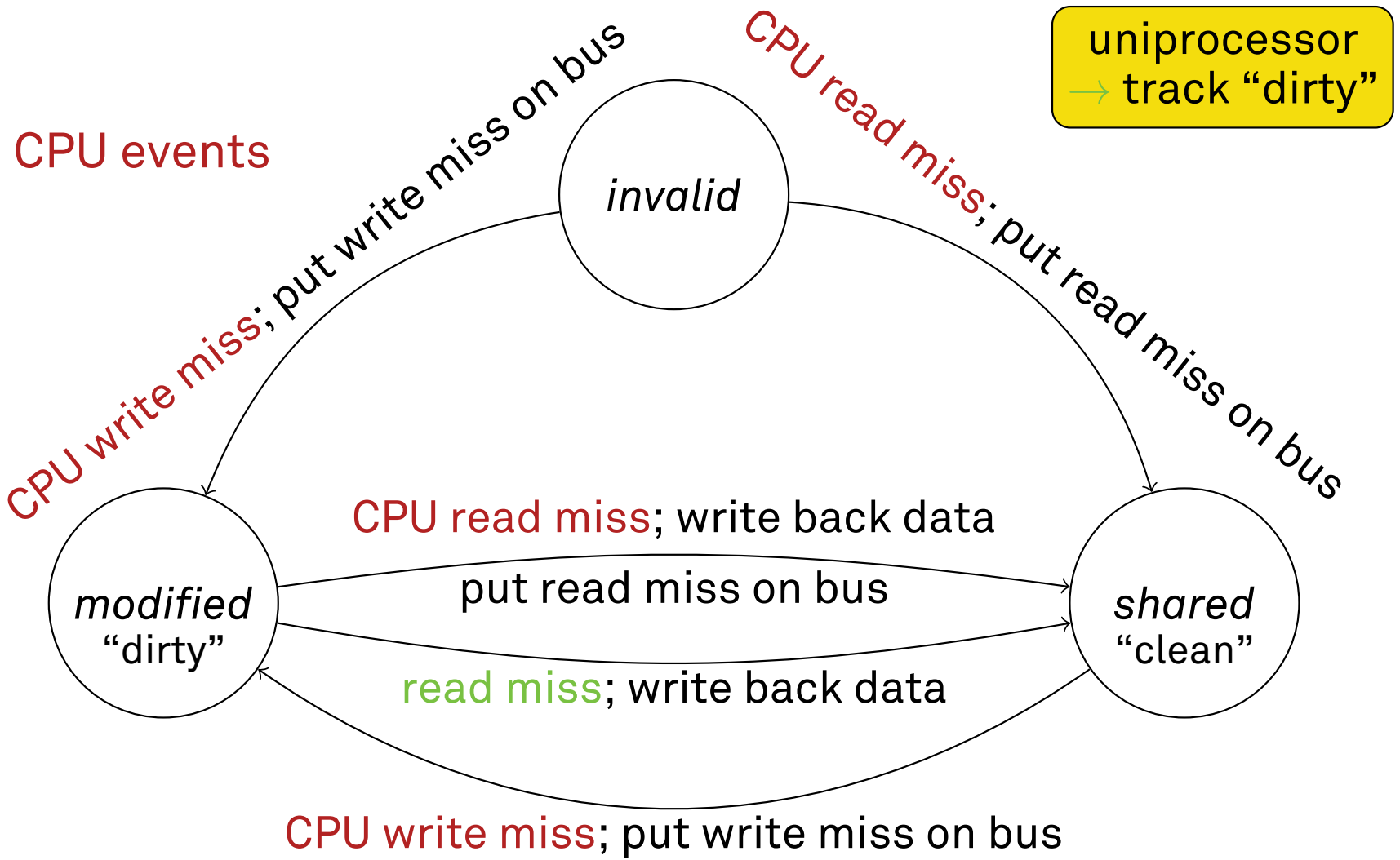
→ Track **sharing state**.

Idea:

- Sending an *invalidate* will make local copy the only one valid.
  → Mark local cache line as *modified* (≈ *exclusive*).
- If a local cache line is already *modified*, writes need **not** be announced on the bus (no *invalidate* message).
- Upon read request by other processor:
  → If local cache line has state *modified*, **answer** the request by sending local version.
  → Change local cache state to *shared*.

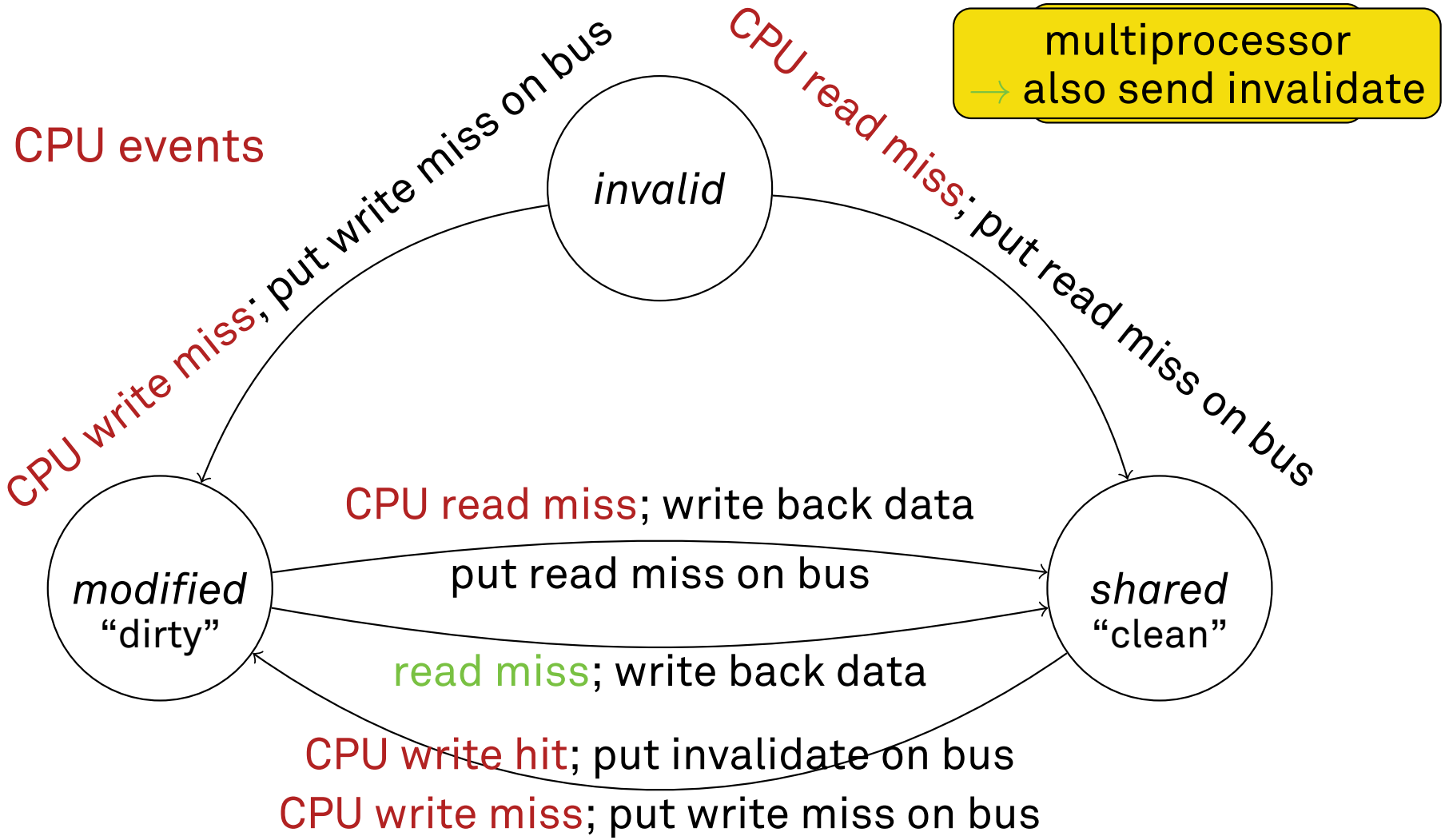
Write Invalidate—State Machine

Local caches track sharing states using a **state machine.**



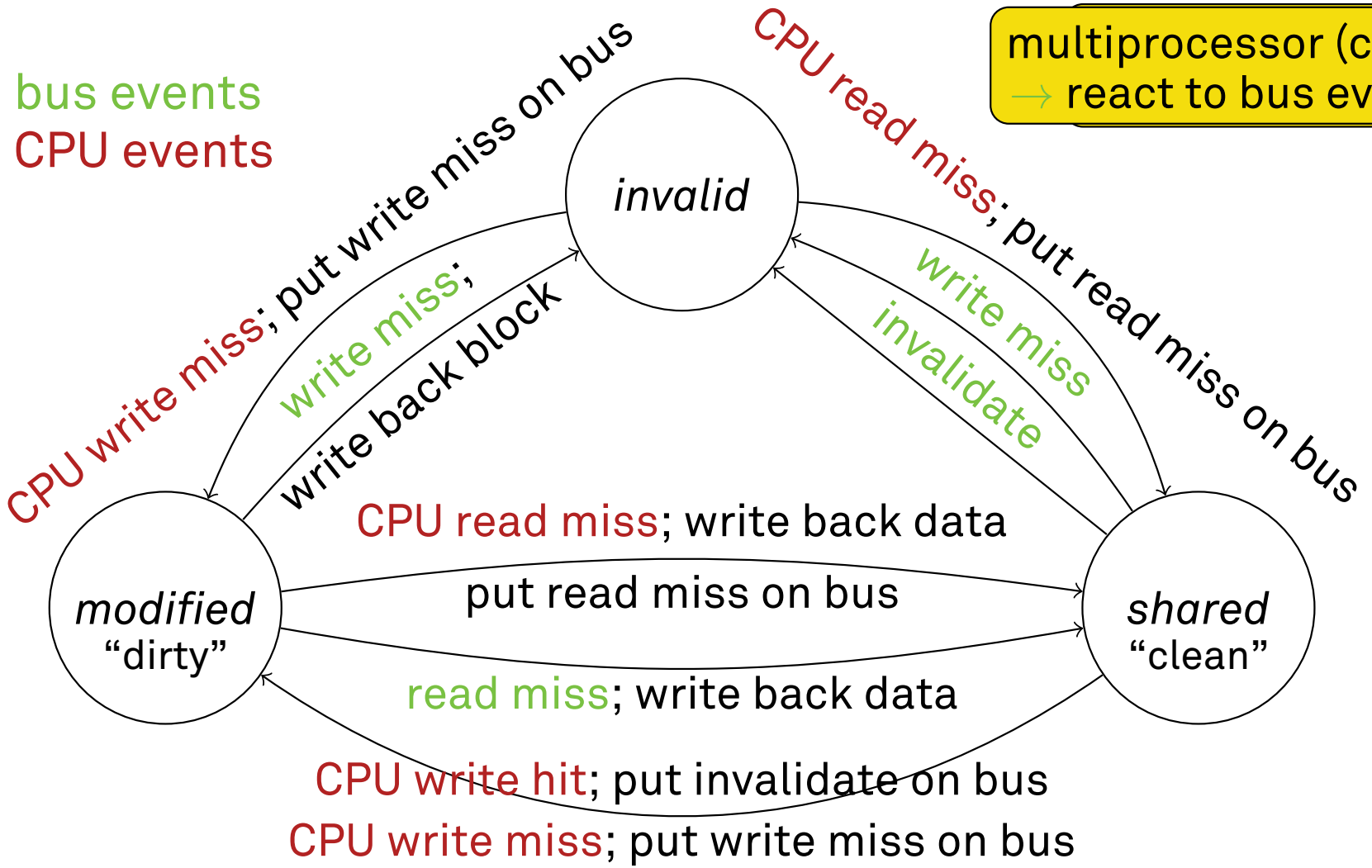
Write Invalidate—State Machine

Local caches track sharing states using a **state machine.**



Write Invalidate—State Machine

Local caches track sharing states using a **state machine.**



Write Invalidate—Notes

Notes:

- Because of the three states *modified*, *shared*, and *invalid*, the protocol on the previous slide is also called ***MSI* protocol.**
- The *Write Invalidate* protocol ensures that any valid cache block is either
  - in the ***shared* state in one or more caches** or
  - in the ***modified* state in exactly one cache.**
    (Any transition to the *modified* state invalidates all other copies of the block; whenever another cache fetches a copy of the block, the *modified* state is left.)
- The *MSI* protocol also ensures that every *shared* item has also been written back to memory. ◁

MSI Protocol—Extensions

Actual systems often use **extensions** to the *MSI* protocol, *e.g.*,

MESI (“*E*” for *exclusive*)

- Distinguish between *exclusive* (but clean) and *modified* (which implies that the copy is exclusive).
- Optimizes the (common) case when an item is first read ($\rightsquigarrow$ *exclusive*) then modified ($\rightsquigarrow$ modified).

MESIF (“*F*” for *forward*)

- In *M(E)SI*, if *shared* items are served by caches (not only by memory), **all** caches might answer miss requests.
- *MESIF* extends the protocol, so at most one *shared* copy of an item is marked as *forward*. Only this cache will respond to misses on the bus.
- Intel i7 employs the *MESIF* protocol.

MSI Protocol—Extensions

MOESI (“*O*” for *owned*)

- *owned* marks an item that might be outdated in memory; the owner cache is “responsible” for the item.
- The owner **must** respond to data requests (since main memory might be outdated).
- *MOESI* allows moving around dirty data between caches.
- The AMD Opteron uses the *MOESI* protocol.
- MOESI avoids the need to write every shared cache block back to memory ($\rightsquigarrow \triangleleft$).

Limitations of a Shared Bus

Limitations of a shared bus:

- Large core counts $\rightarrow$ **high bandwidth.**
- Shared buses cannot satisfy bandwidth demands of modern multiprocessor systems.

Therefore:

- **Distribute** memory
- Communicate through **interconnection network**

Consequence:

- **Non-uniform memory access (NUMA)** characteristics

Bandwidth Demand

***E.g.*, Intel Xeon E7-8880 v3:**

- 2.3 GHz clock rate
- 18 cores per chip (36 threads)
- Up to 8 processors per system

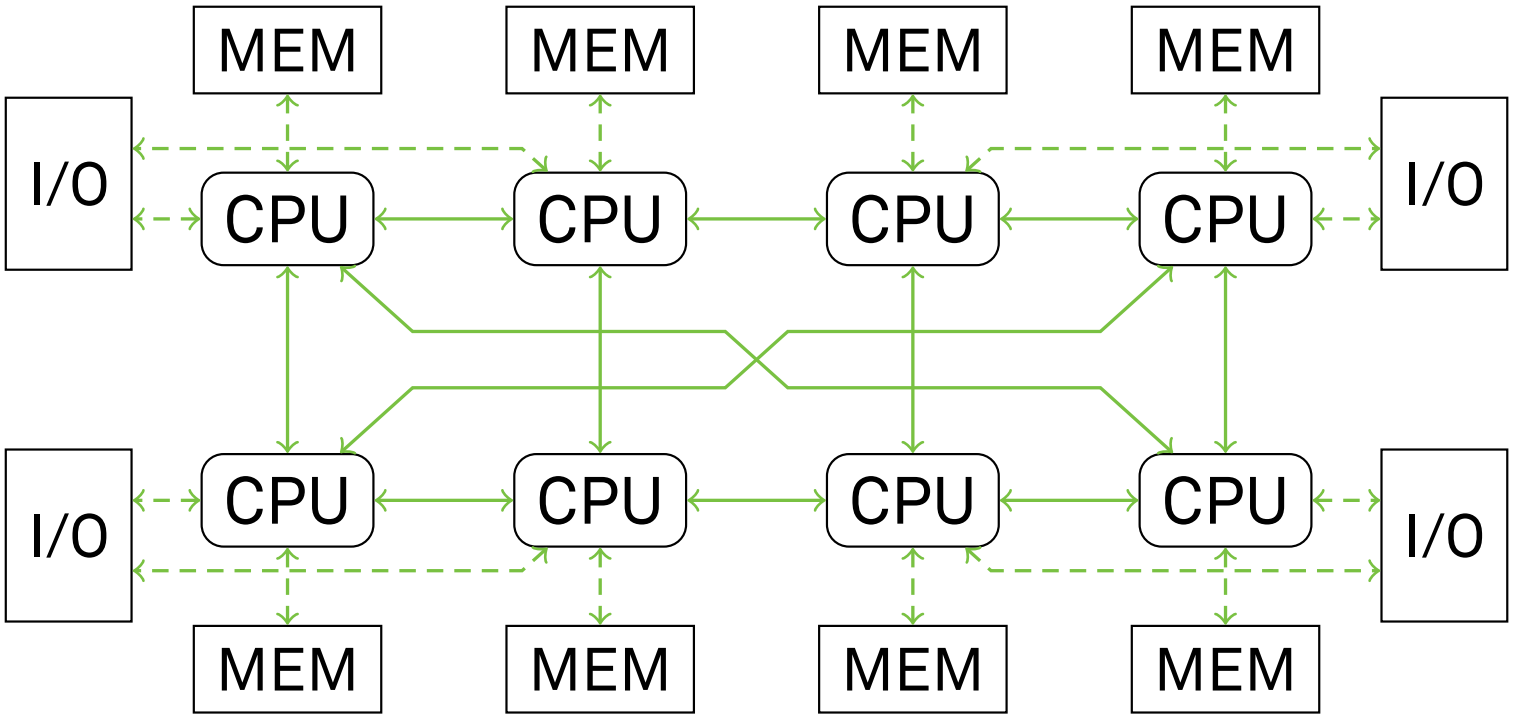
Back-of-the-envelope calculation:

- 1 byte per cycle per core $\rightarrow$ 331 GB/s
- Data-intensive applications might demand much more!

Shared memory bus?

- Modern bus standards can deliver at most a few ten GB/s.
- Switching very high bandwidths is a challenge.

Example: 8-Way Intel Nehalem-EX



- Interconnect: “Intel Quick Path Interconnect (QPI)”[8]
- Memory may be local, one hop away, or two hops away.
  - → Non-uniform memory access (NUMA)

[8] The AMD counterpart is “HyperTransport”.

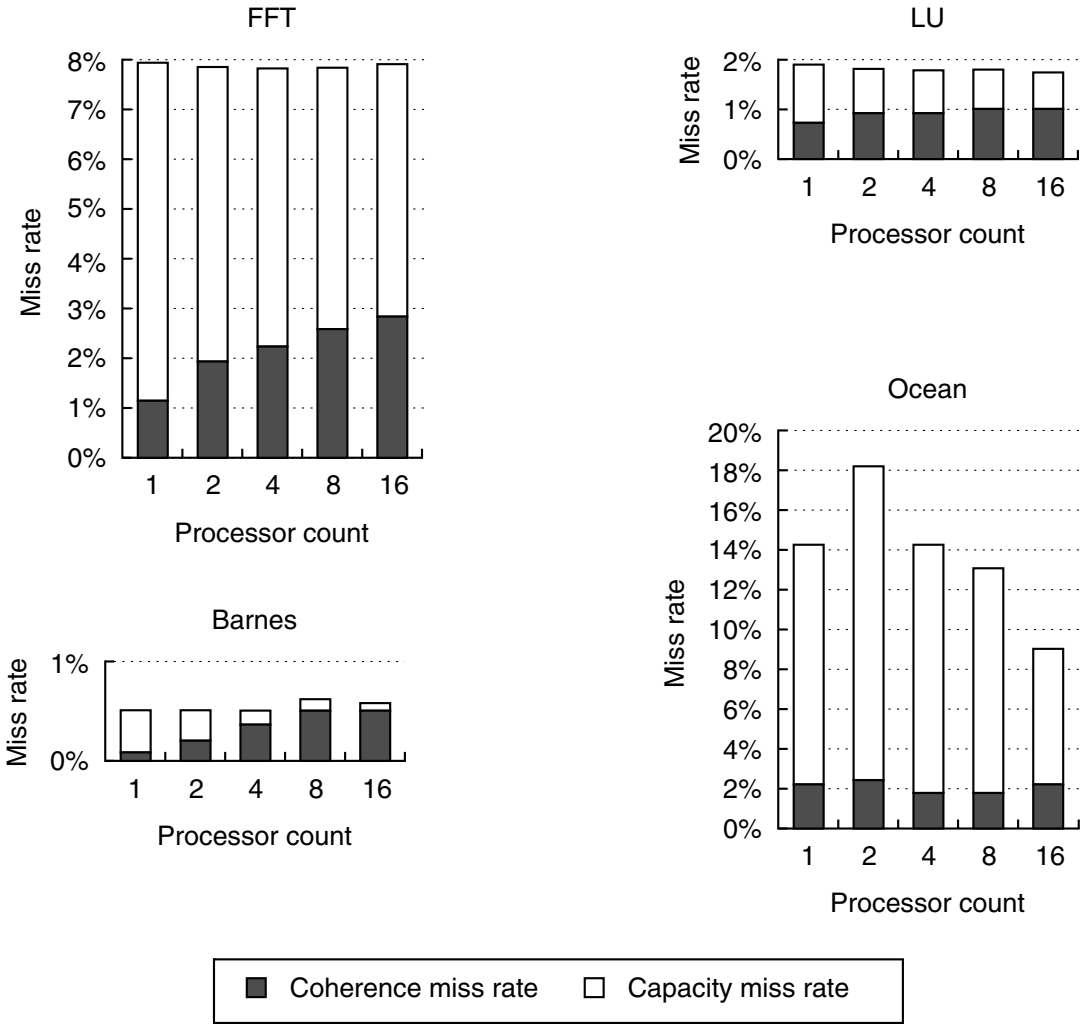
Distributed Memory and Snooping

Idea:

- Extend “snooping” to distributed memory.
- **Broadcast** coherence traffic, send data **point-to-point.**

✎ **Problem solved?**

Snooping-Based Cache Coherency: Scalability



Example:

- Scientific Applications
- ↗ Hennessy & Patterson, Sect. I.5

→ AMD Opteron is a system that still uses the approach.

Directory-Based Cache Coherence

To avoid all-broadcast coherence protocol:

- Use a **directory** to keep track of which item is replicated where.
- Direct coherence messages only to those nodes that actually need them.

Directory:

- Either keep a **global directory** ($\rightsquigarrow$ scalability?).
- Or define a **home node** for each memory address.
  - $\rightarrow$ Home node holds directory for that item.
  - $\rightarrow$ Typically: distribute directory along with memory.

Protocol now involves

- **directory/-ies** (at item home node(s)),
- **individual caches** (local to processors).

Parties communicate **point-to-point** (no broadcasts).

Directory-Based Cache Coherence

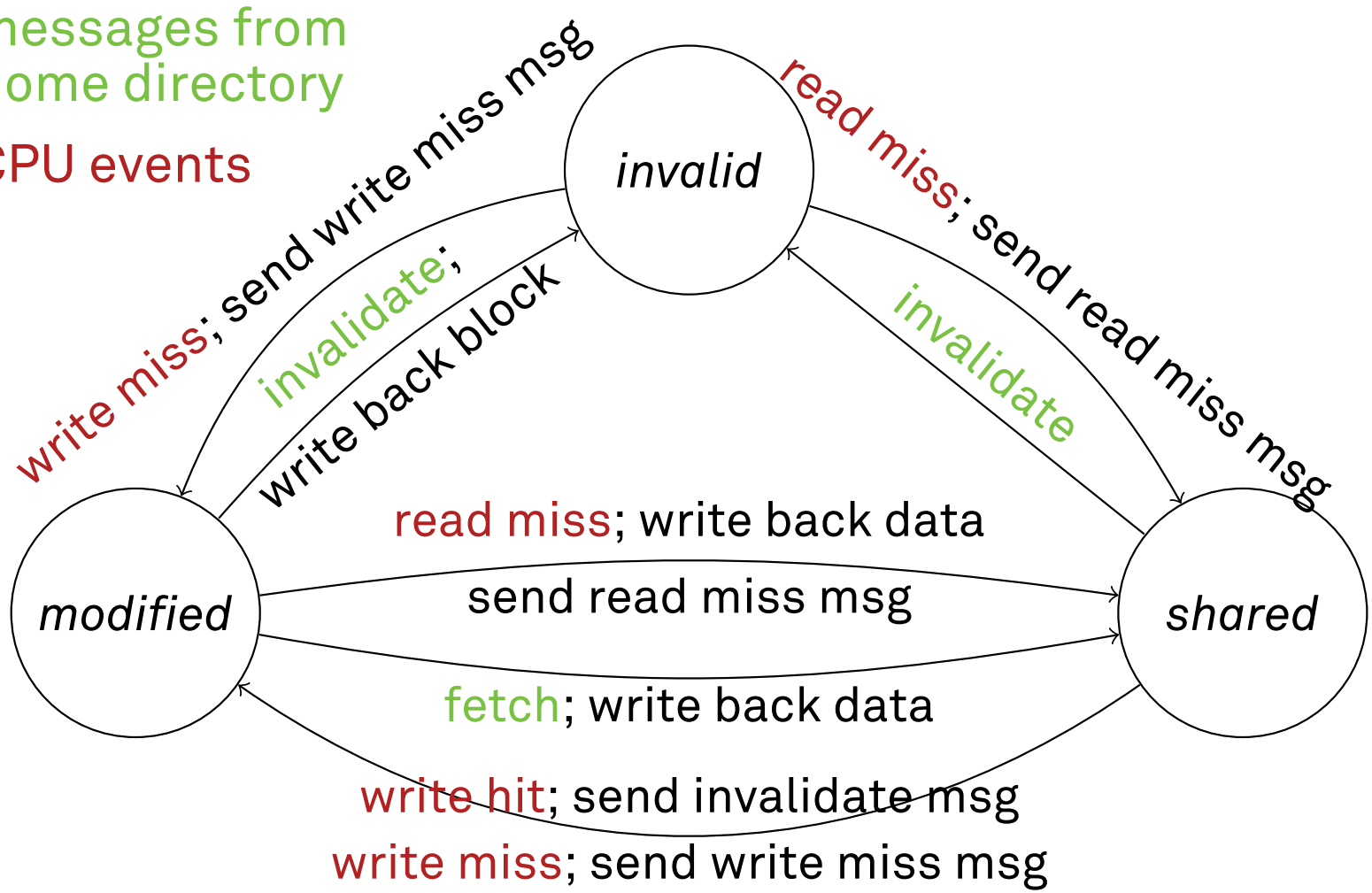
Messages sent by individual nodes:

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

↗ Hennessy & Patterson, Computer Architecture, 5th edition, page 381.

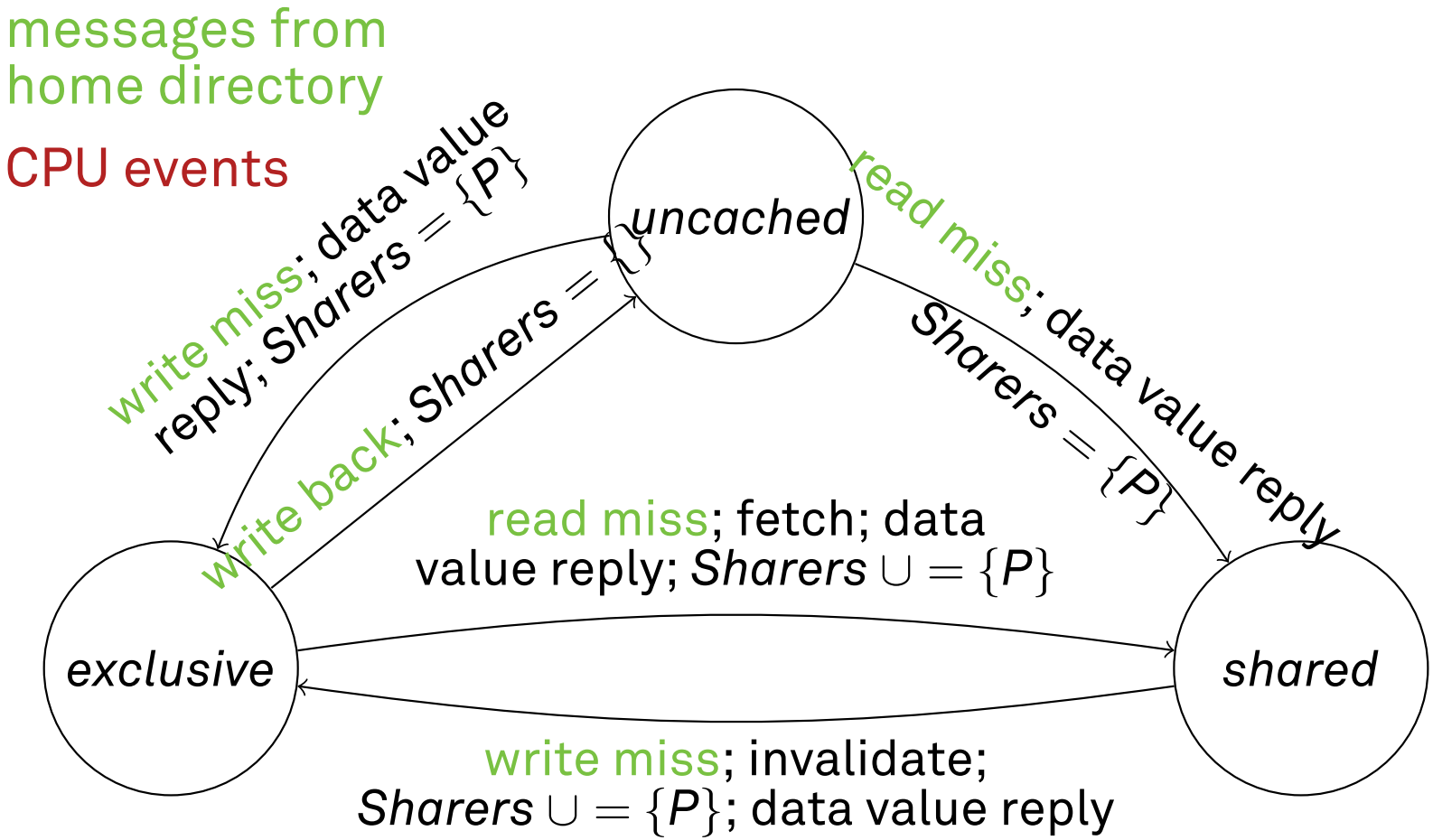
Directory-Based Coherence—State Machine

Individual caches use a state machine similar to the one on slide 95.



Directory-Based Coherence—State Machine

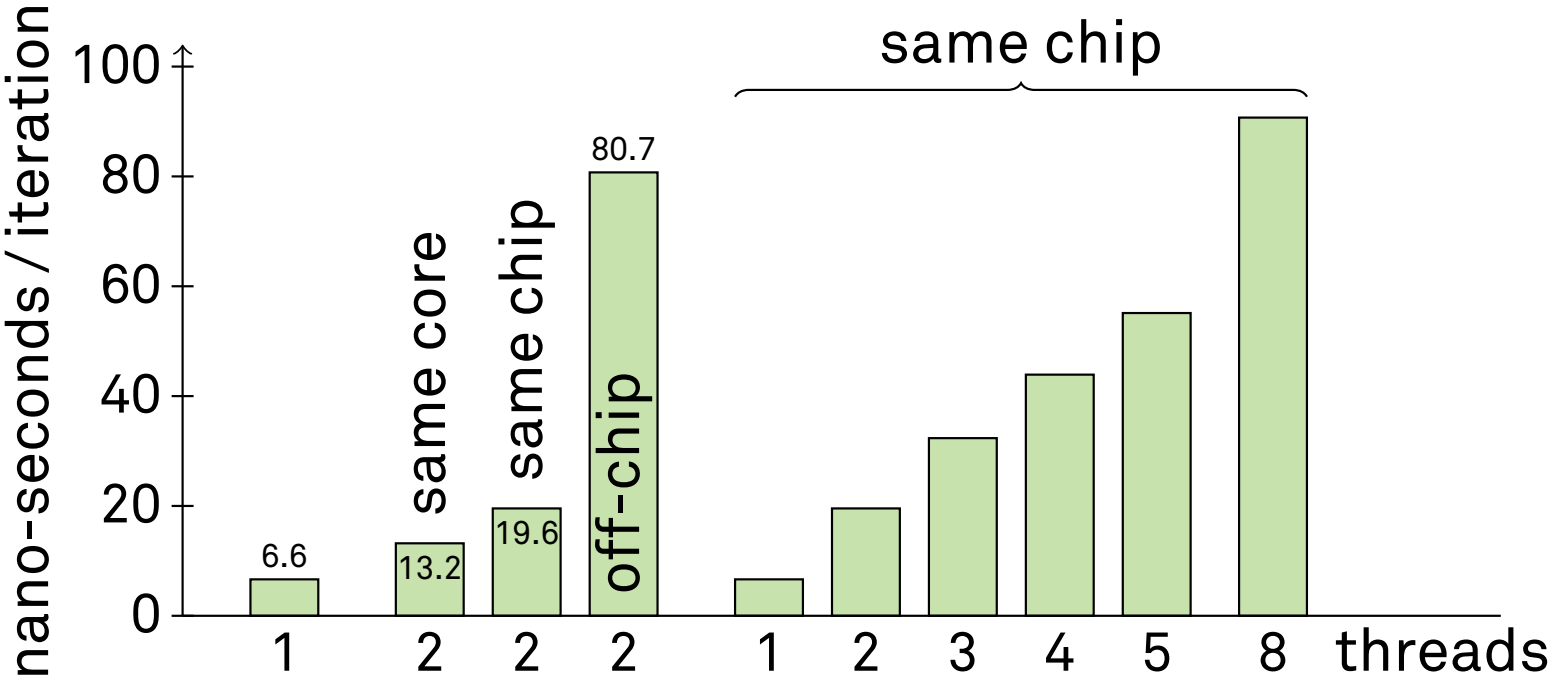
The **directory** has its own state machine.



Cache Coherence Cost

Experiment:

- Several threads randomly increment elements of an integer array; Zipfian probability distribution, no synchronization[9].



Intel Nehalem EX; 1.87 GHz; 2 CPUs, 8 cores/CPU.

[9] In general, this will yield incorrect counter values.

Cache Coherence Cost

Two types of **coherence misses**:

true sharing miss

→ Data shared among processors.

→ Often-used mechanism to **communicate** between threads.

→ These misses are **unavoidable**.

false sharing miss

→ Processors use **different data items**, but the items reside in the **same cache line.**

→ Items get invalidated/migrated, even though no data is actually shared.

✎ **How can false sharing misses be avoided?**

Synchronization—Producer/Consumer

Shared data is a convenient mechanism to **communicate** between threads.

✎ **Producer/consumer protocol based on (only) shared memory?**

Shared Memory Consistency

Programs like the producer/consumer scenario make assumptions about the observed **order** of memory operations.

Another example:

Thread T_1:

```
   /* prepare data          */
1  data ← 0x4711;
   /* tell T2 we're done    */
2  done ← 1;
```

Thread T_2:

```
   /* wait for T1           */
1  while done = 0 do
       /* do nothing        */
   /* use data              */
2  print (data);
```

→ Cache coherence will **not** ensure correct behavior here!

✎ **Why?**

Shared Memory Consistency

Need **consistency model** to reason about program behavior.

Most straightforward model: **sequential consistency**

> *A multiprocessor system is **sequentially consistent** 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.*

Sequential Consistency

Example:

- Assume two programs:
  1. Program A: $\langle A_1, A_2, A_3 \rangle$
  2. Program B: $\langle B_1, B_2, B_3 \rangle$
- Sequential consistency allows program behavior as if any of the sequences
  - $\langle A_1, A_2, B_1, A_3, B_2, B_3 \rangle$
  - $\langle A_1, B_1, B_2, A_2, A_3, B_3 \rangle$
  - $\langle B_1, B_2, A_1, A_2, A_3, B_3 \rangle$
  - ...

  were executed.

  However, program runs equivalent to
  - $\langle A_1, B_2, A_2, B_1, A_3, B_3 \rangle$ or
  - $\langle B_1, B_2, A_3, A_2, B_3, A_1 \rangle$

  are **not** allowed.

Sequential Consistency

Where's the problem?

- Sequential consistency: Result as if all memory accesses by each processors were kept in order.
- But:
  - **Write buffers** make write operations asynchronous.
  - Individual memory operations might experience different **delays** in the **interconnect network**.
  - **Dynamic scheduling** and **speculation** also alter the order of execution.

Solution? 

- Enforce in-order execution for memory operations?
  - → *E.g.*, wait for **acknowledgements** from all caches for *invalidate* messages.

Relaxed Consistency Models

Relaxations of sequential consistency sacrifice consistency guarantees to achieve better performance.

Idea: Relax the strict handling of

- $W \rightarrow R$ (write followed by a read) program order,
- etc.

Ask programmers/compilers to explicitly **mark** situations where consistency must be maintained.

E.g.,

- Relax $W \rightarrow R$ ("total store ordering" or "processor consistency"):
  - Many programs will depend on the order among writes. These will run just fine.
  - Can still hide write latency.
  - ✎ **example on slide 111?**

Relaxed Consistency Models

- Relax $W \rightarrow R$ and $W \rightarrow W$ (“partial store order”)
  - Writes to different locations may appear in different order.
  - Allows combined writes.
- “weak ordering”
  - No order guarantees.
  - Goes well with dynamic scheduling.
  - Programs must ask for ordering explicitly.

Synchronization in Relaxed Consistency Models

With relaxed consistency, applications may have to provide explicit **synchronization** instructions.

E.g.,

- **fence operations**
  - Wait until all outstanding memory operations have been completed.
  - *E.g.*, `LFENCE`, `SFENCE`, `MFENCE` on Intel x86
- **“locked” instructions**
  - Intel allows some instructions to be prefixed with `LOCK`. This allows for complex instructions to appear atomic.
  - Locked instructions are not re-ordered.

Synchronization—Mutual Exclusion

Two threads want to use a critical region. Synchronization based on (only) shared memory?

Need for Atomic Operations

Problem:

- A lock implementation for n threads that only uses shared memory must write to $\mathcal{O}(n)$ distinct memory locations.
- With unclear memory consistency, even that might not be sufficient.

Therefore:

- Hardware support for **atomic read/modify operations.**

E.g.,

- `test-and-set` (v, a): Set variable v to a and return the value that v had before the assignment.
- `fetch-and-increment` (v): Return the value of v and atomially increment it.

Atomic Operations and Instruction Set

Instruction set for atomic operations:

- Either explicit “test and set,” “fetch and increment,” etc. instructions.
  - → *E.g.*, `XADD` (Exchange and Add) on Intel x86.
- Or **pairs of instructions** that operate together
  - → “load linked”/“load locked” and “store conditional”

E.g., MIPS **atomic exchange** (↗ Hennessy & Patterson)

```
try: MOV  R3,R4      ;mov exchange value
     LL   R2,0(R1)   ;load linked
     SC   R3,0(R1)   ;store conditional
     BEQZ R3,try     ;branch if store fails
     MOV  R4,R2      ;put load value in R4
```

Implementing Atomic Operations

Another example: **fetch and increment**

```
try: LL      R2,0(R1)   ;load linked
     DADDUI  R3,R2,#1   ;increment
     SC      R3,0(R1)   ;store conditional
     BEQZ    R3,try     ;branch if store fails
```

Implementation:

- On `LL`, remember address in **link register**.
- **Clear** link register whenever
  - an interrupt occurs or
  - an *invalidate* is received for the address in the link register.
- On `SC`, **compare address to link register.**

Using Atomic Operations—Spinlock

✎ **Implementation of a spinlock?**

See the (excellent) article of Tom Anderson (“The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors.” IEEE Trans. on Parallel and Distributed Systems, vol. 1(1), 1990).

Locking Strategies

There are two strategies to implement locking:

Spinning ("busy waiting"; can be done in user space)

- Waiting thread repeatedly **polls** lock until it becomes free.
- Cost: two **cache miss penalties** (if implemented well)

  $\rightarrow \approx 150$ nsec
- Thread burns CPU cycles while spinning.

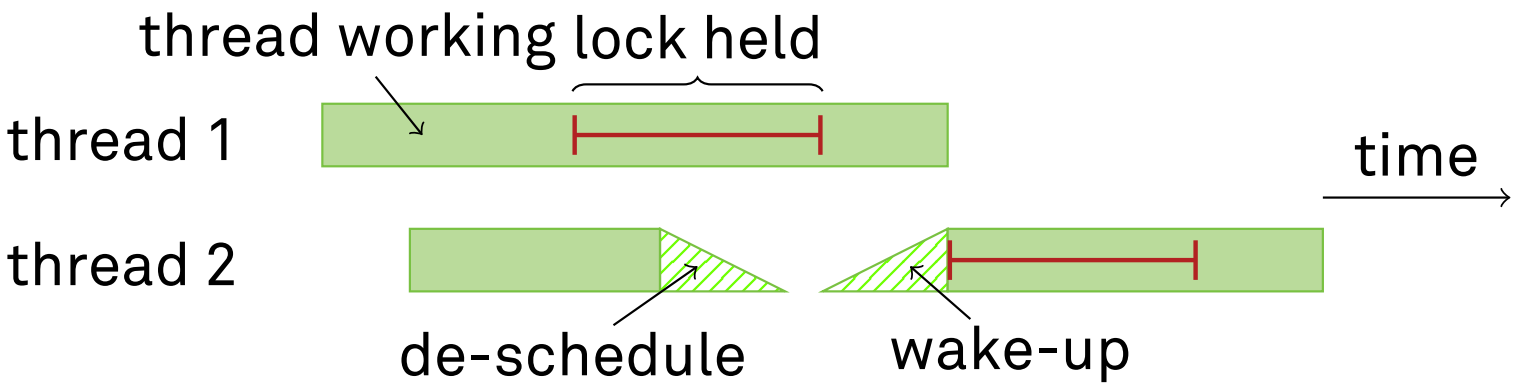
Blocking (operating system service)

- **De-schedule** waiting thread until lock becomes free.
- Cost: two **context switches** (one to sleep, one to wake up)

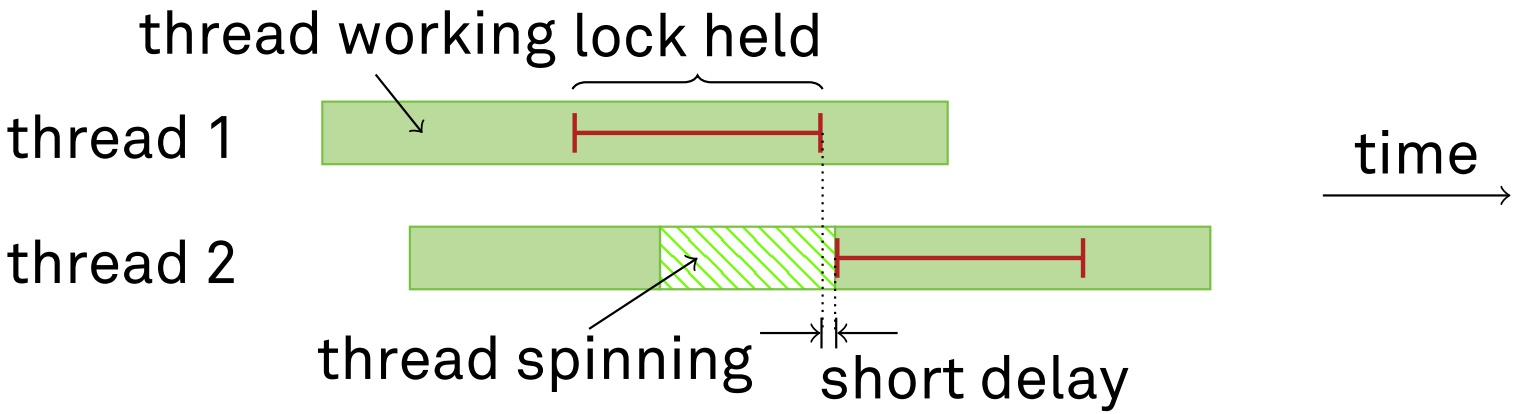
  $\rightarrow \approx 12$–$20\ \mu$sec

Thread Synchronization

Blocking:

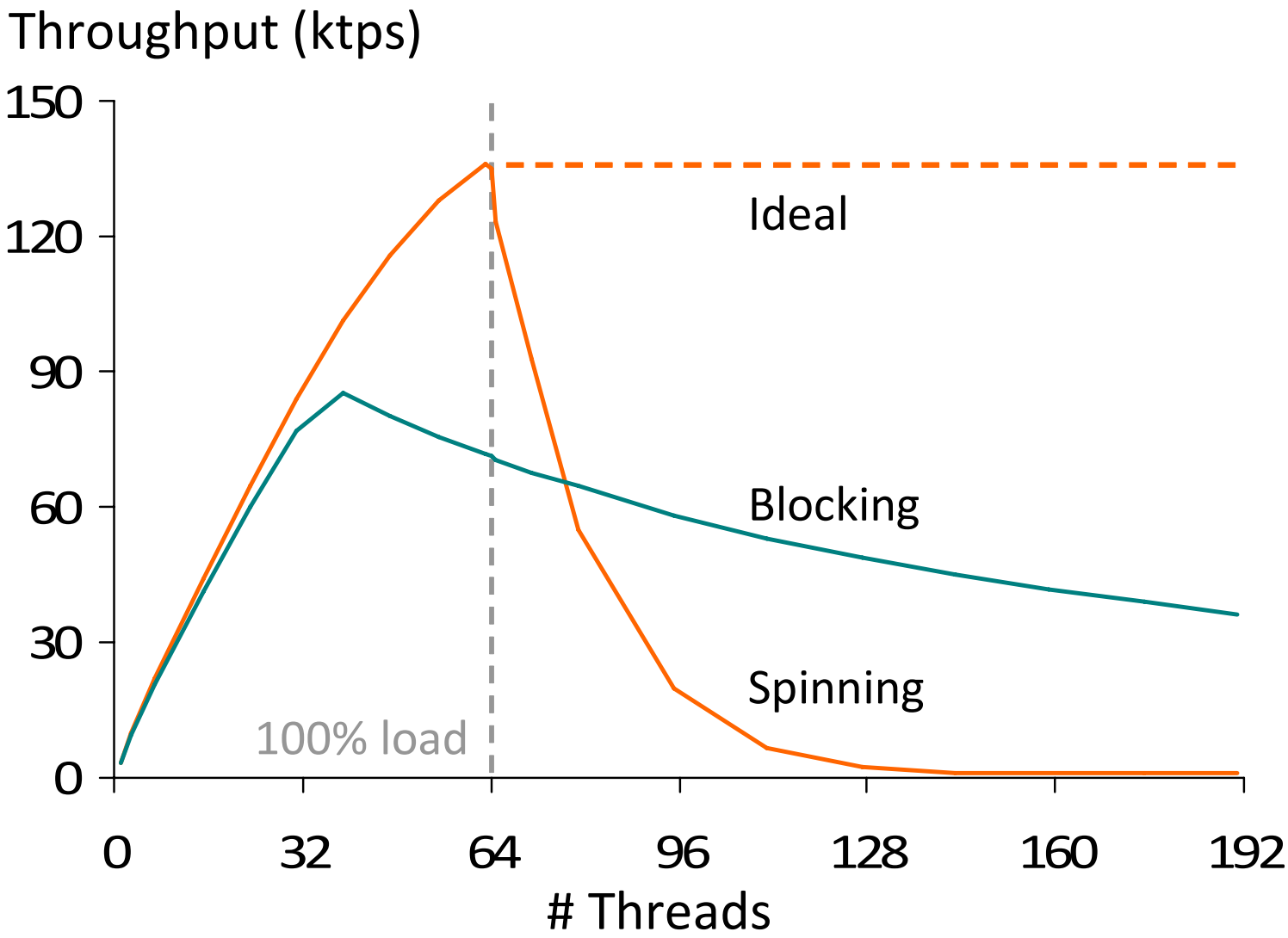


Spinning:



Experiments: Locking Performance

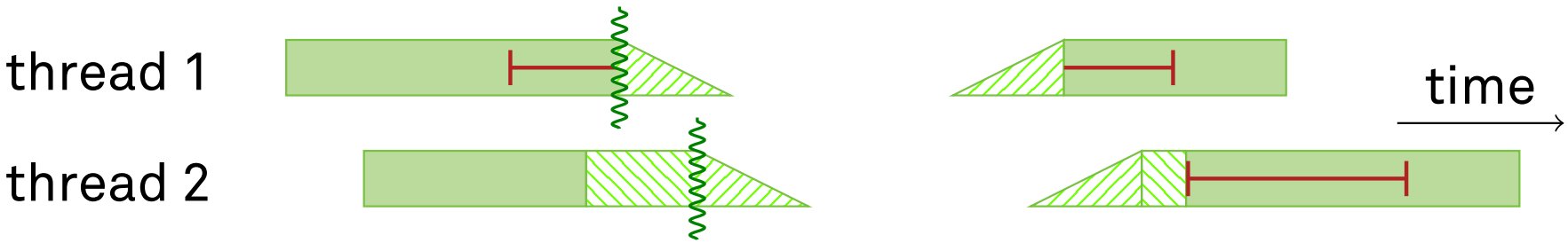
Sun Niagara II (64 hardware contexts):



Source: Johnson *et al.* Decoupling Contention Management from Scheduling. *ASPLOS 2010.*

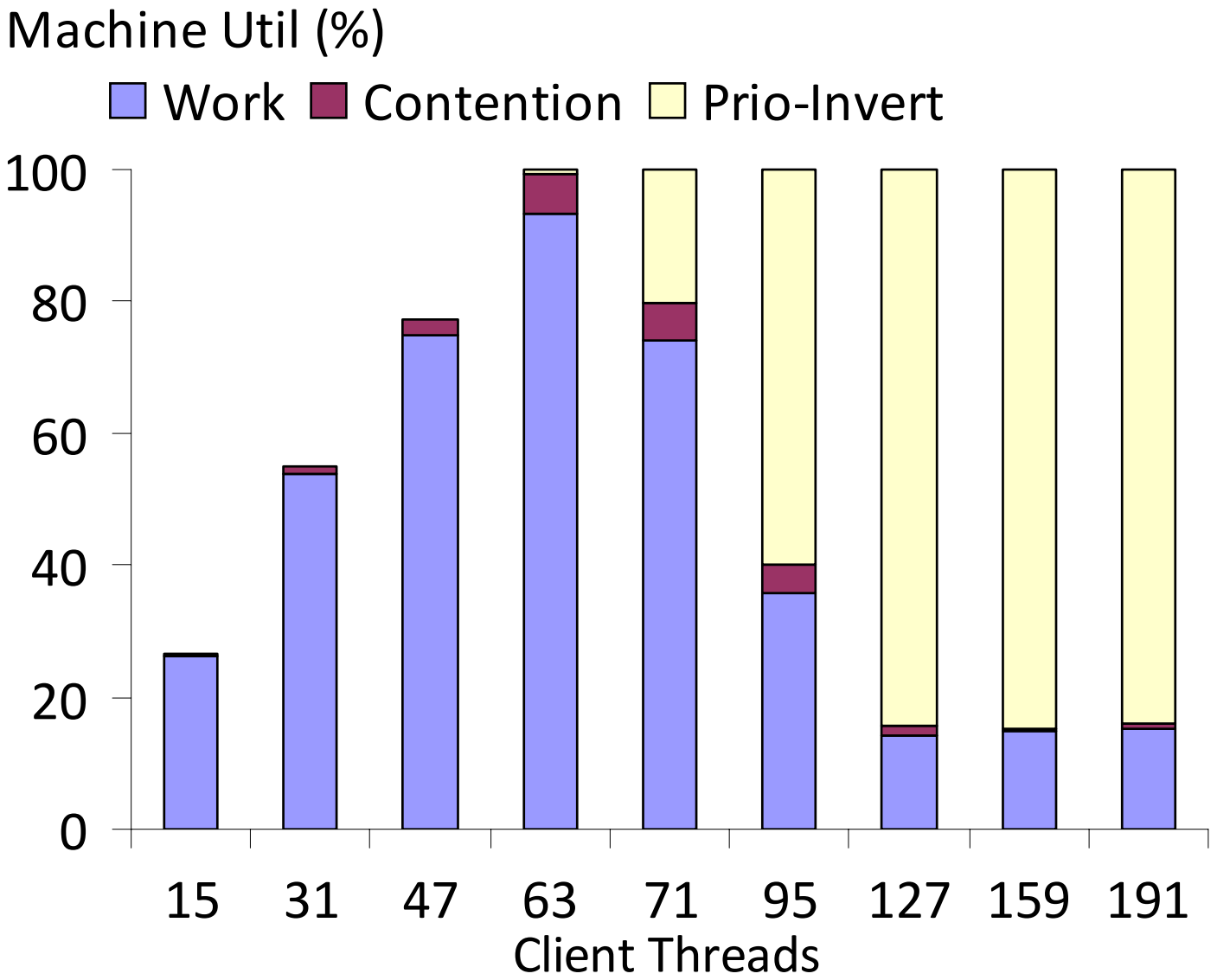
Spinning Under High Load

Under **high load**, spinning can cause problems:



- More threads than hardware contexts.
- Operating system **preempts** running task .
  - Working and spinning threads all appear busy to the OS.
  - Working thread likely had longest time share already
    → gets **de-scheduled** by OS.
- **Long** delay before working thread gets re-scheduled.
- By the time working thread gets re-scheduled (and can now make progress), waiting thread likely gets de-scheduled, too.

Spinning



Source: Johnson *et al.* Decoupling Contention Management from Scheduling. *ASPLOS 2010.*