Data Processing on Modern Hardware

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

Winter 2018/19

Part V

Execution on Multiple Cores

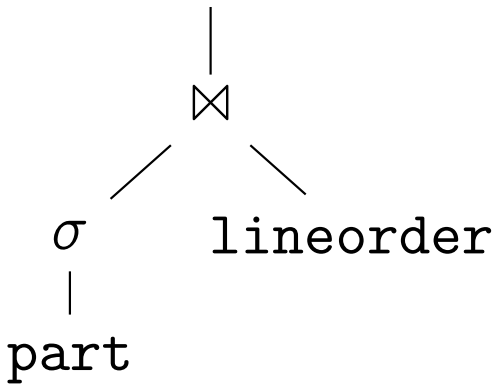
Example: Star Joins

Task: run parallel instances of the query (↗ introduction)

dimension

fact table

```
SELECT SUM(lo_revenue)
  FROM part, lineorder
 WHERE p_partkey = lo_partkey
   AND p_category <= 5
```

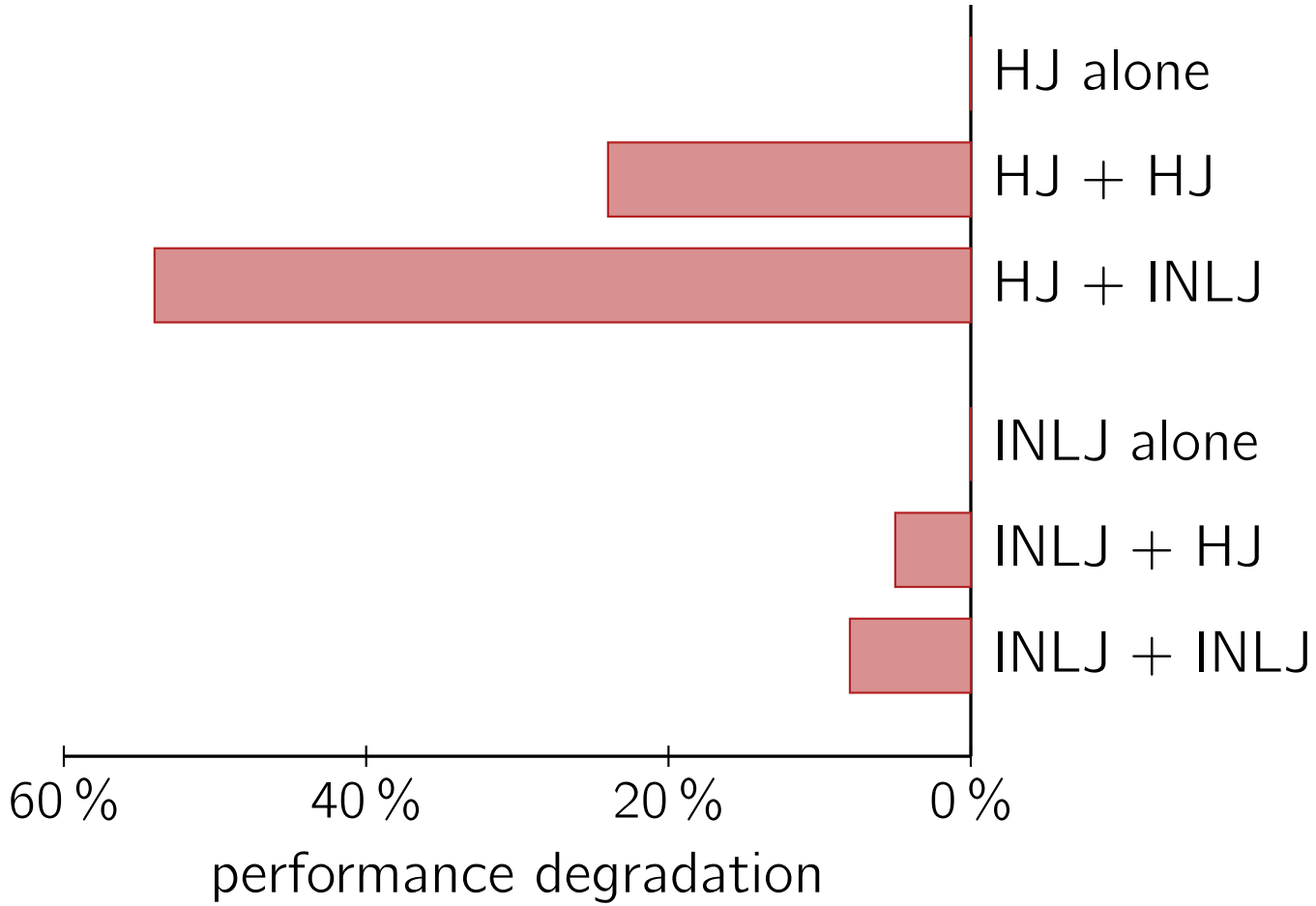


To implement ⋈ use either

- a **hash join** or
- an **index nested loops join**.

Execution on "Independent" CPU Cores

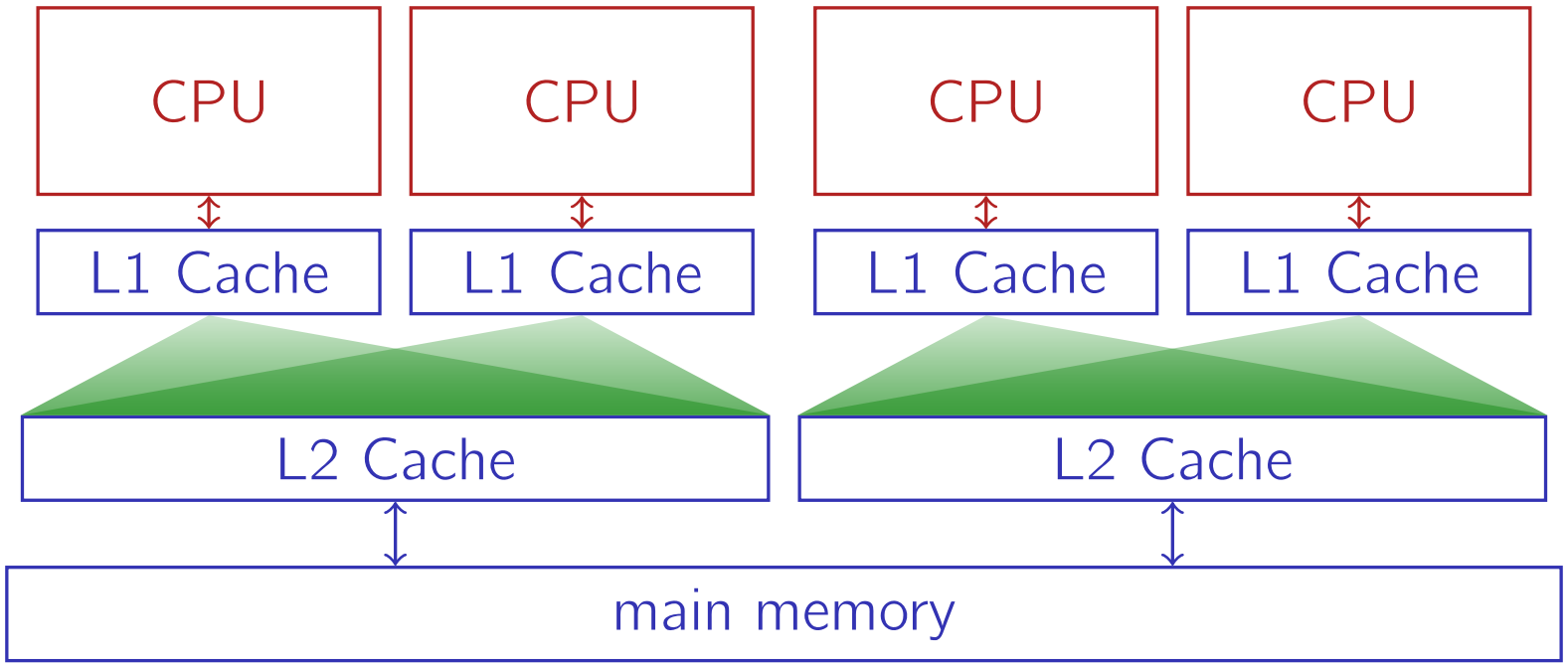
Co-run independent instances on different CPU cores.



Concurrent queries may seriously affect each other's performance.

Shared Caches

In Intel Core 2 Quad systems, two cores **share** an L2 Cache:

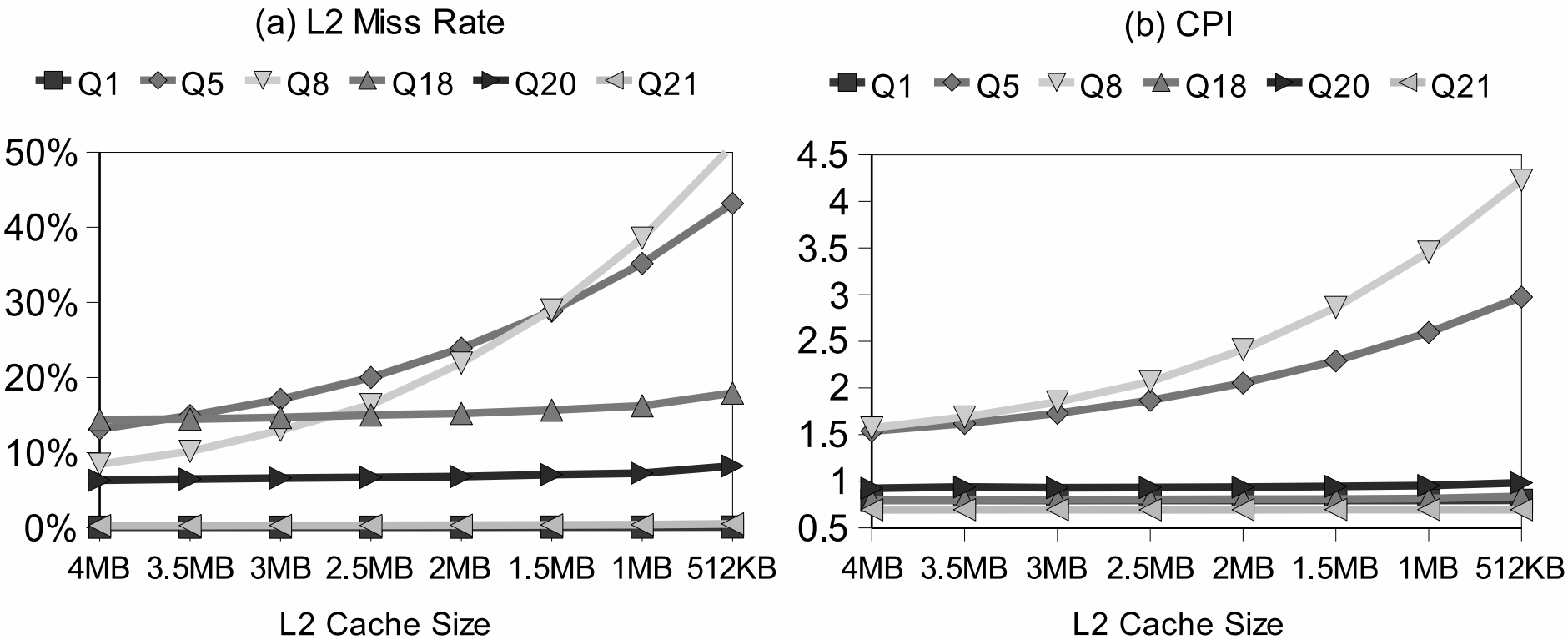


What we saw was **cache pollution**.

→ How can we avoid this cache pollution?

Cache Sensitivity

Dependence on cache sizes for some TPC-H queries:



Some queries are more sensitive to cache sizes than others.

- **cache sensitive:** hash joins
- **cache insensitive:** index nested loops joins; hash joins with very small or very large hash table

Locality Strength

This behavior is related to the **locality strength** of execution plans:

Strong Locality

small data structure; reused very frequently

- *e.g.*, small hash table

Moderate Locality

frequently reused data structure; data structure $\approx$ cache size

- *e.g.*, moderate-sized hash table

Weak Locality

data not reused frequently or data structure $\gg$ cache size

- *e.g.*, large hash table; index lookups

Execution Plan Characteristics

Locality effects how caches are used:

cache pollution

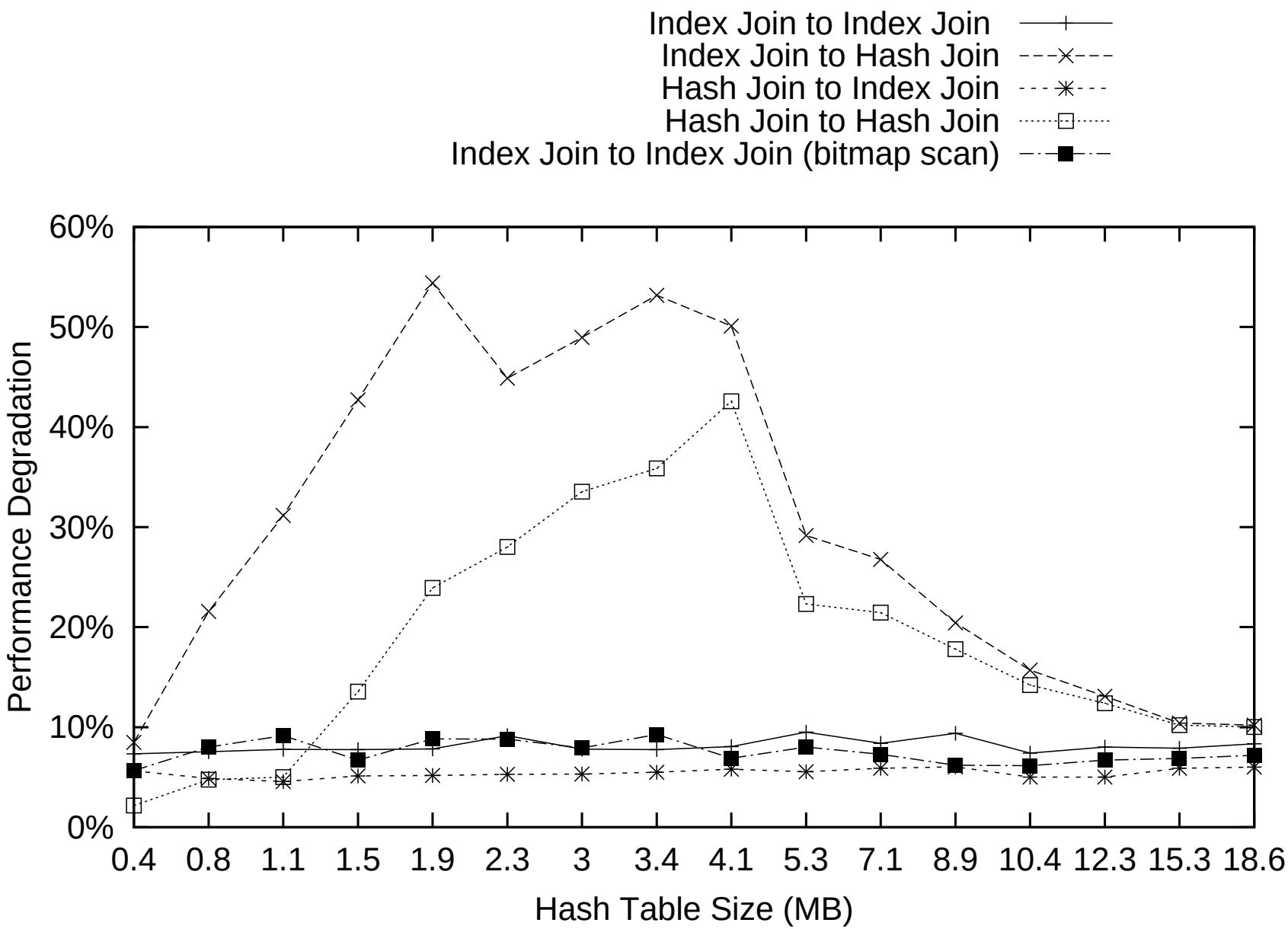
	strong	moderate	weak
amount of cache used	small	large	large
amount of cache needed	small	large	small

Plans with **weak locality** have most severe impact on co-running queries.

Impact of co-runner on query:

	strong	moderate	weak
strong	low	moderate	high
moderate	moderate	high	high
weak	low	low	low

Experiments: Locality Strength



Source: Lee *et al.* MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *VLDB 2009*.

Locality-Aware Scheduling

An optimizer could use knowledge about localities to **schedule** queries.

- **Estimate** locality during query analysis.
  - Index nested loops join $\rightarrow$ weak locality
  - Hash join:

$$\begin{aligned} \text{hash table} \ll \text{cache size} &\rightarrow \text{strong locality} \\ \text{hash table} \approx \text{cache size} &\rightarrow \text{moderate locality} \\ \text{hash table} \gg \text{cache size} &\rightarrow \text{weak locality} \end{aligned}$$

- **Co-schedule** queries to minimize (the impact of) cache pollution.

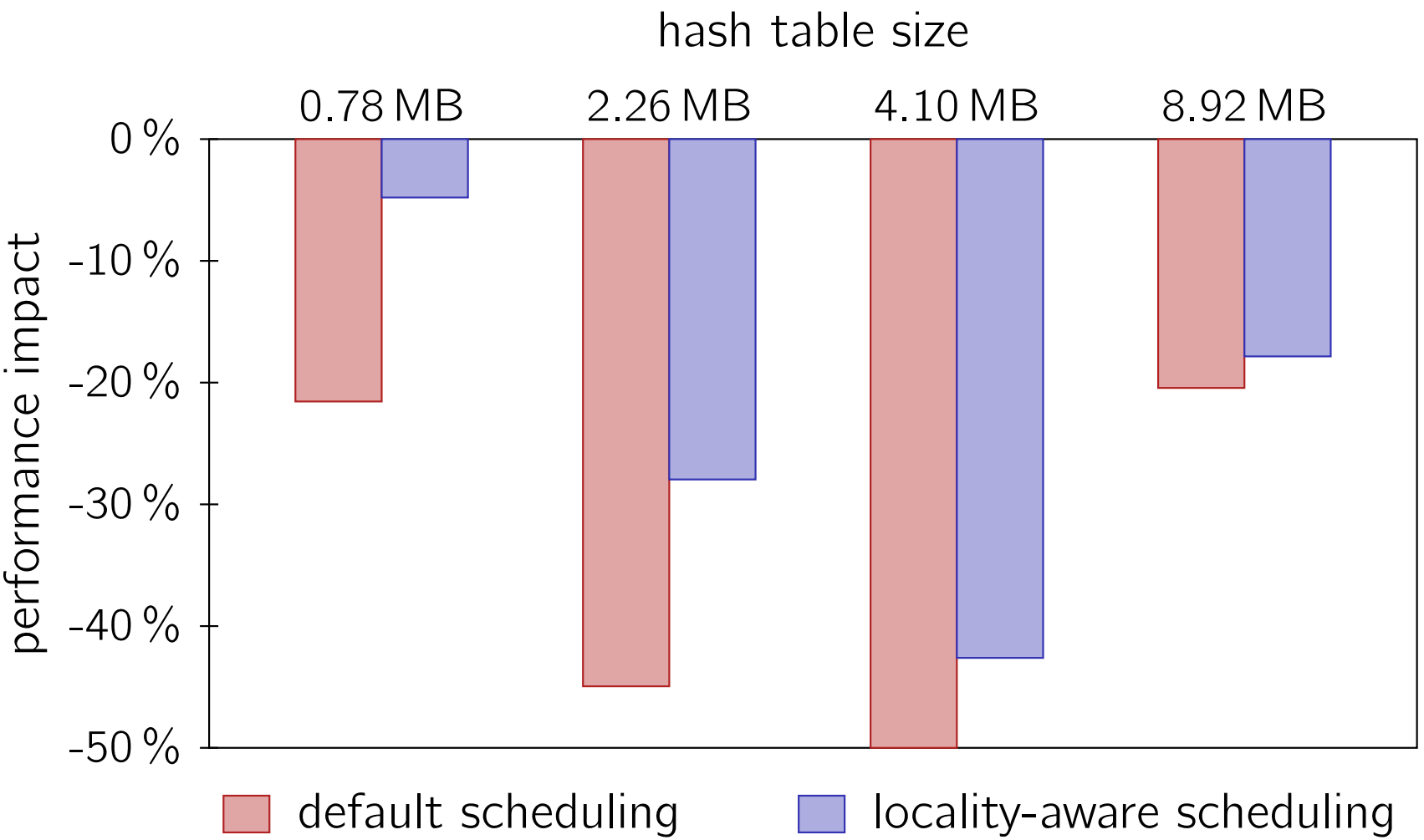
✎ **Which queries should be co-scheduled, which ones not?**

- Only run weak-locality queries next to weak-locality queries.

  $\rightarrow$ They cause high pollution, but are not affected by pollution.
- Try to co-schedule queries with small hash tables.

Experiments: Locality-Aware Scheduling

PostgreSQL; 4 queries (different `p_categorys`); for each query: $2 \times$ hash join plan, $2 \times$ INLJ plan; impact reported for hash joins:

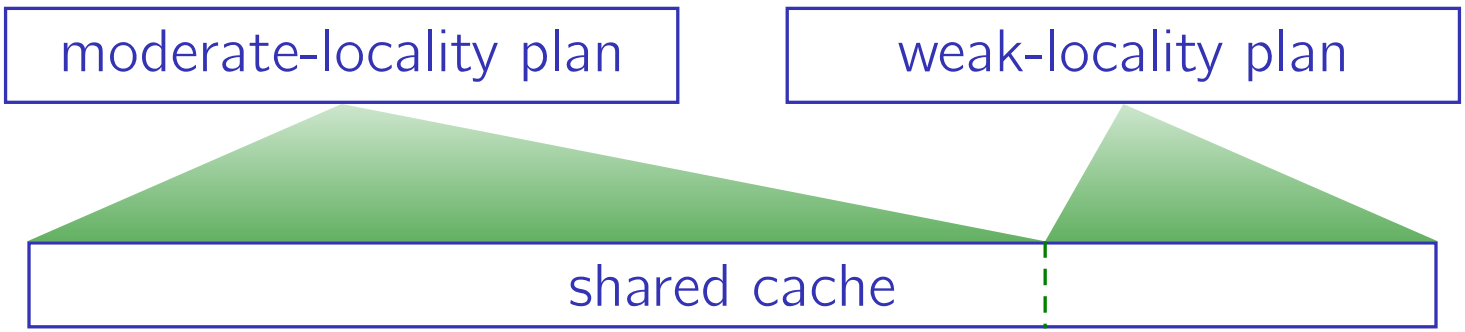


Source: Lee *et al. VLDB 2009.*

Cache Pollution

Weak-locality plans cause cache pollution, because they **use** much cache space even though they do not strictly **need** it.

By **partitioning** the cache we could reduce pollution with little impact on the weak-locality plan.



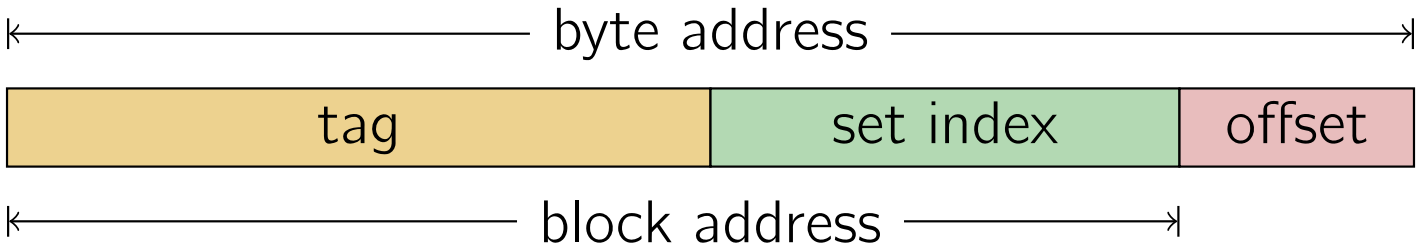
But:

- Cache allocation controlled by **hardware**.

Cache Organization

Remember how caches are organized:

- The **physical address** of a memory block determines the **cache set** into which it could be loaded.



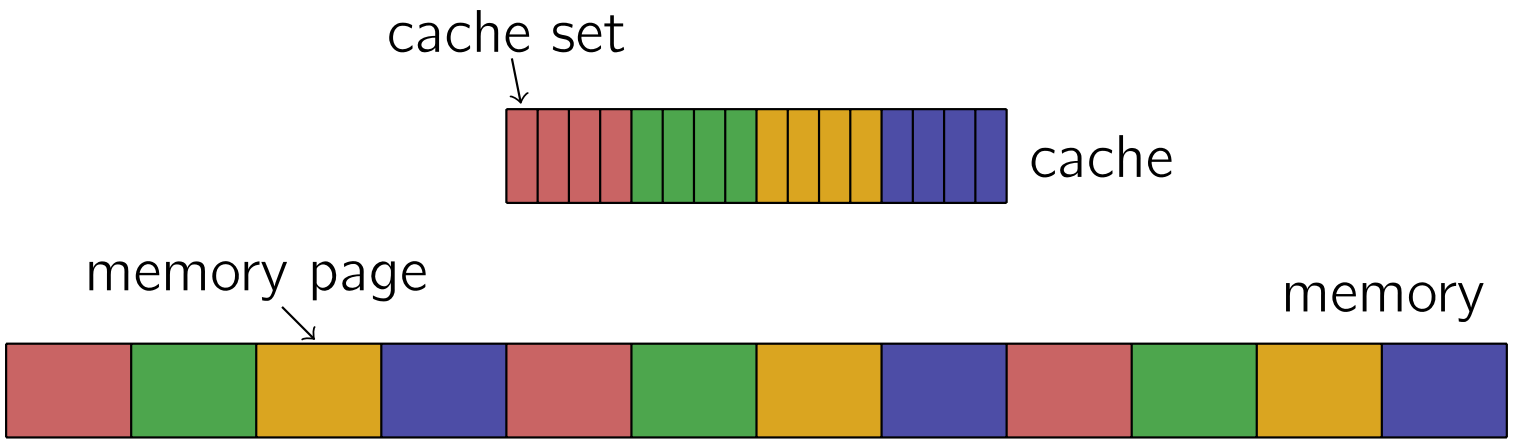
Thus,

- We can **influence hardware behavior** by the **choice of physical memory allocation**.

Page Coloring

The address $\leftrightarrow$ cache set relationship inspired the idea of **page colors**.

- Each memory page is assigned a **color**.[5]
- Pages that map to the **same cache sets** get the **same color**.



✎ **How many colors are there in a typical system?**

[5] Memory is organized in **pages**. A typical **page size** is **4 kB**.

Page Coloring

- By using memory only of certain colors, we can effectively restrict the cache region that a query plan uses.

Note that

- Applications (usually) have **no control** over physical memory.
- Memory allocation and virtual $\leftrightarrow$ physical mapping are handled by the **operating system**.
- We need **OS support** to achieve our desired **cache partitioning**.

MCC-DB: Kernel-Assisted Cache Sharing

MCC-DB ("Minimizing Cache Conflicts"):

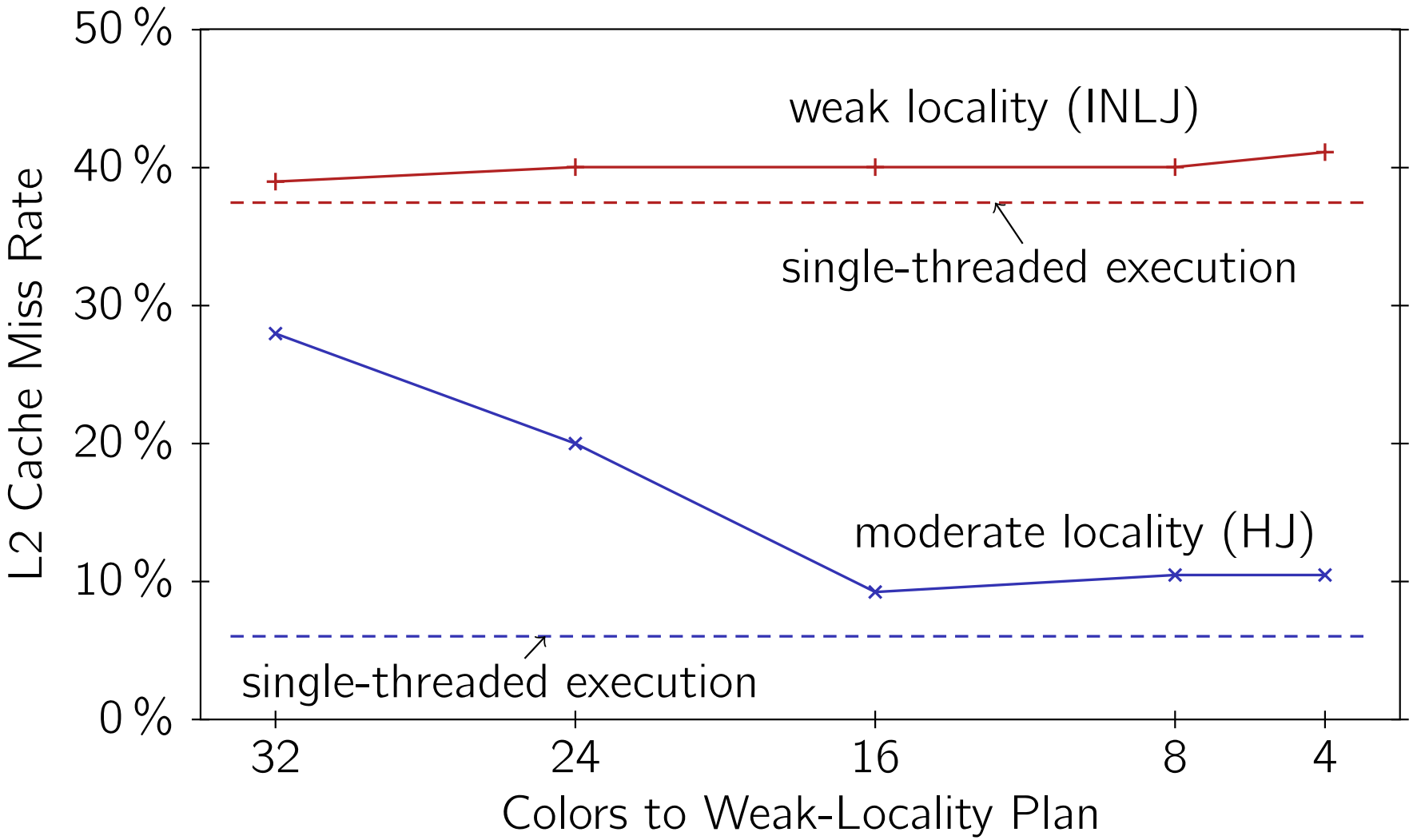
- Modified Linux 2.6.20 kernel
  - Support for **32 page colors** (4 MB L2 Cache: 128 kB per color)
  - **Color specification** file for each process (may be modified by application at any time)
- Modified instance of PostgreSQL
  - **Four colors** for regular buffer pool

    ✎ **Implications on buffer pool size (16 GB main memory)?**

  - For **strong- and moderate-locality** queries, allocate colors as needed (*i.e.*, as estimated by query optimizer)

Experiments

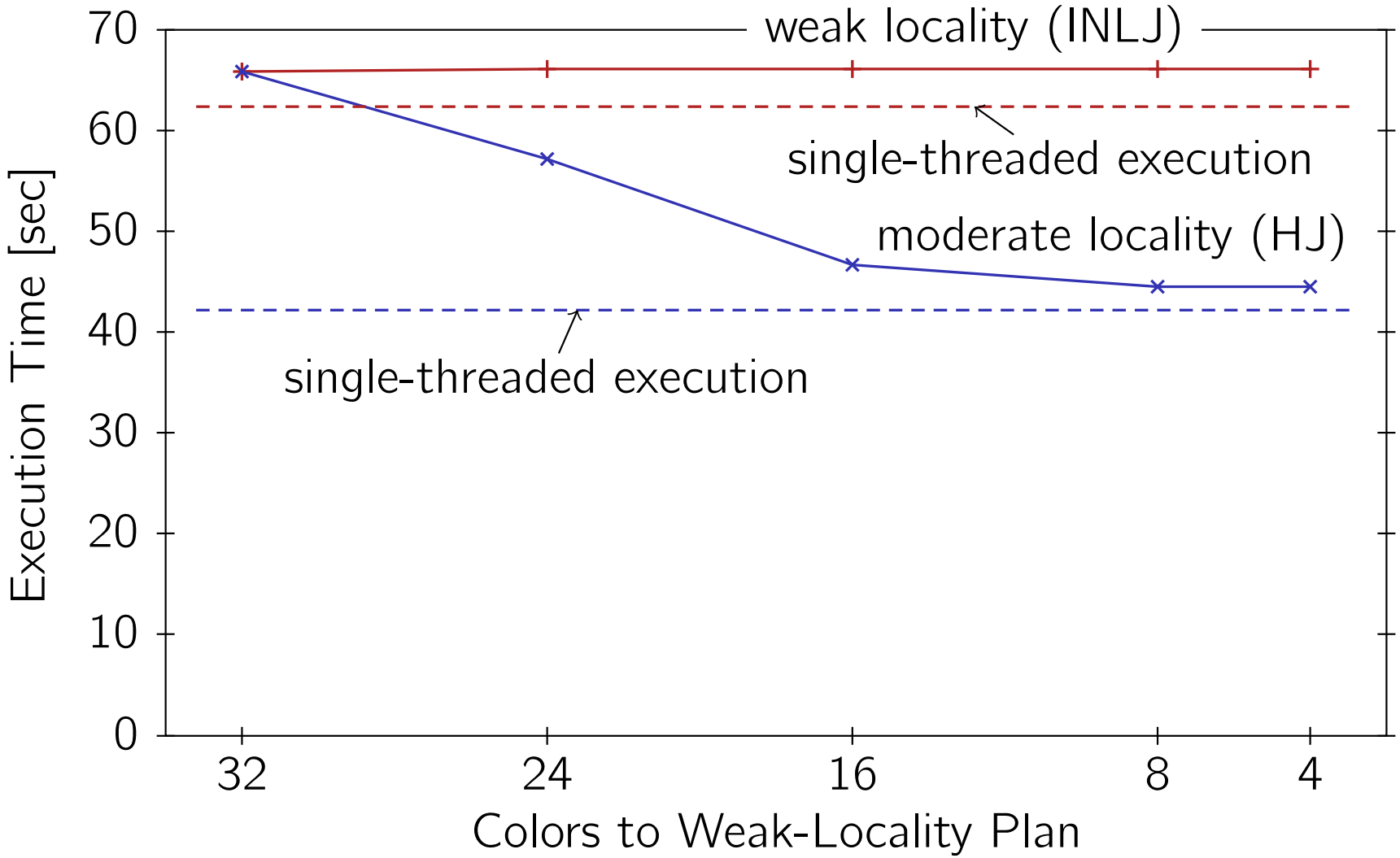
Moderate-locality hash join and weak-locality co-runner (INLJ):



Source: Lee *et al.* *VLDB 2009.*

Experiments

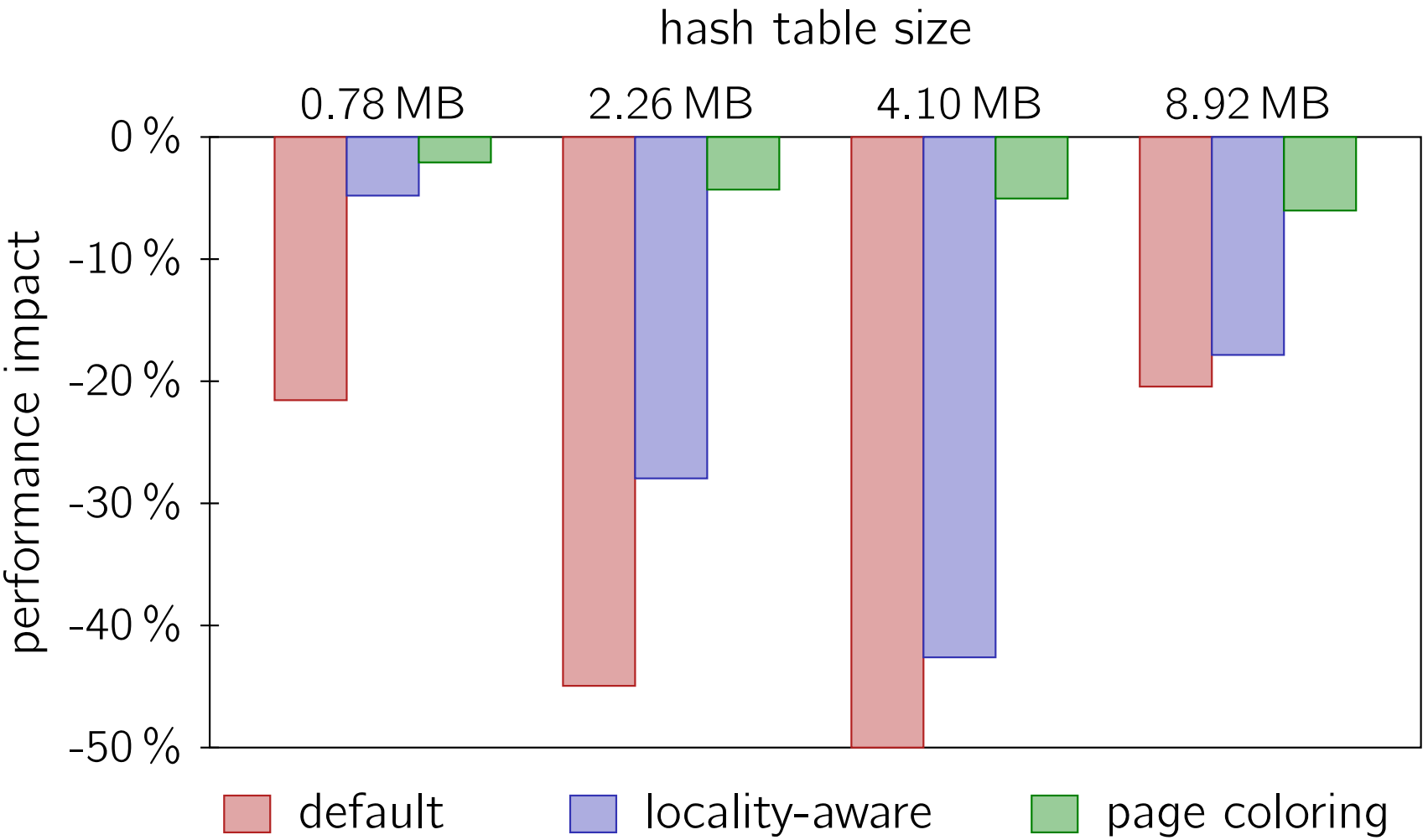
Moderate-locality hash join and weak-locality co-runner (INLJ):



Source: Lee *et al.* *VLDB 2009.*

Experiments: MCC-DB

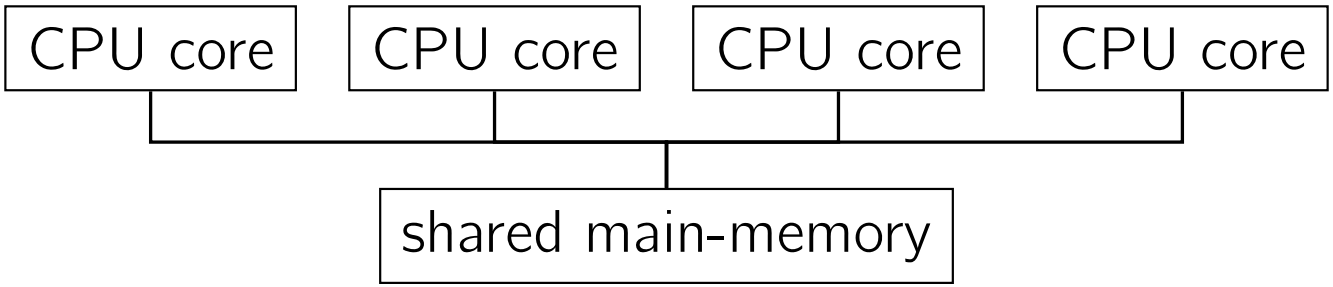
PostgreSQL; 4 queries (different `p_categorys`); for each query: 2 × hash join plan, 2 × INLJ plan; impact reported for hash joins:



Source: Lee *et al. VLDB 2009.*

Building a Shared-Memory Multiprocessor

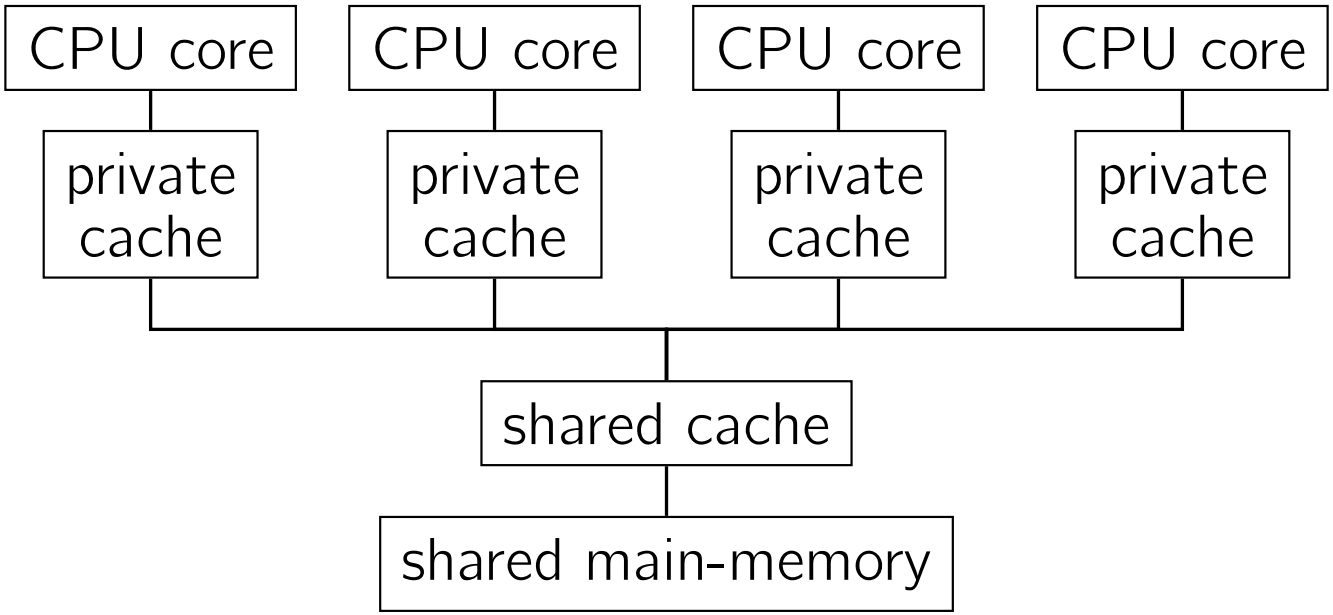
What the programmer likes to think of...



✎ **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:

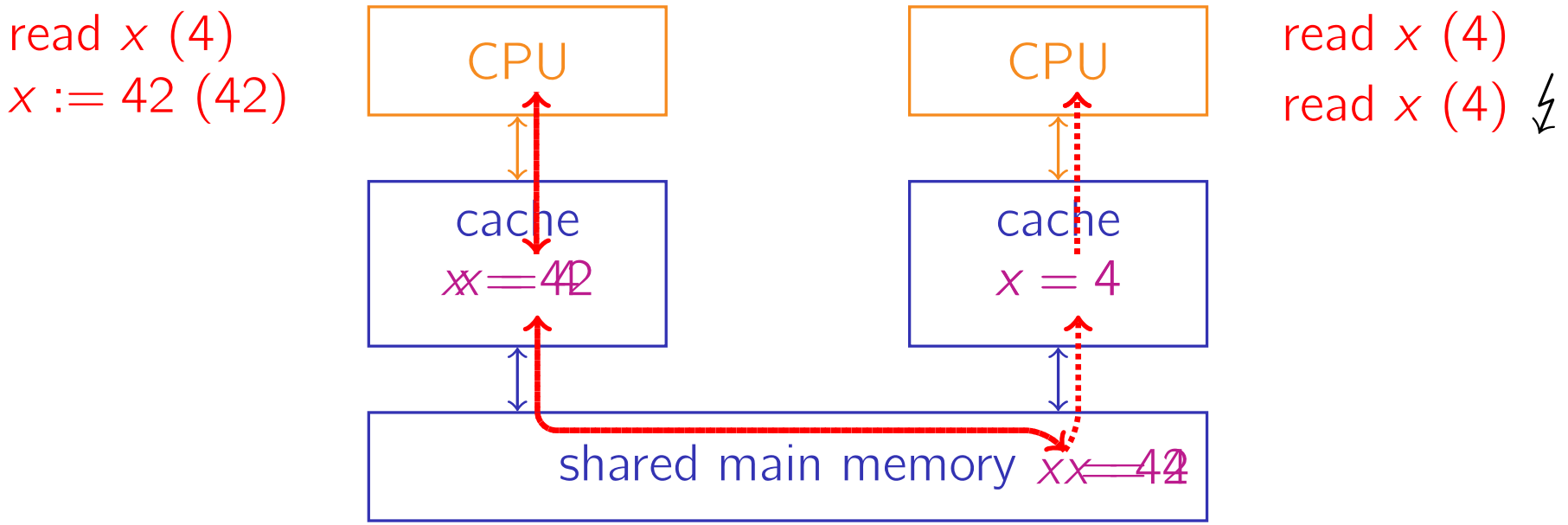
+ **symmetric design**; uniform access time for every memory item from every processor
+ **private data** gets **cached locally**
  → behavior identical to that of a uniprocessor
? **shared data** will be **replicated to private caches**
  → Okay for parallel **reads**.
  → But what about **writes** 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:[6]

> *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?**

[6] 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**.[7]

✎ **Pros/Cons of this strategy?**

[7] 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	~~$x = 4$~~	$x = 42$	$x = 4$[8]
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 (⇝ necessary for write-back caches).

[8] 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.

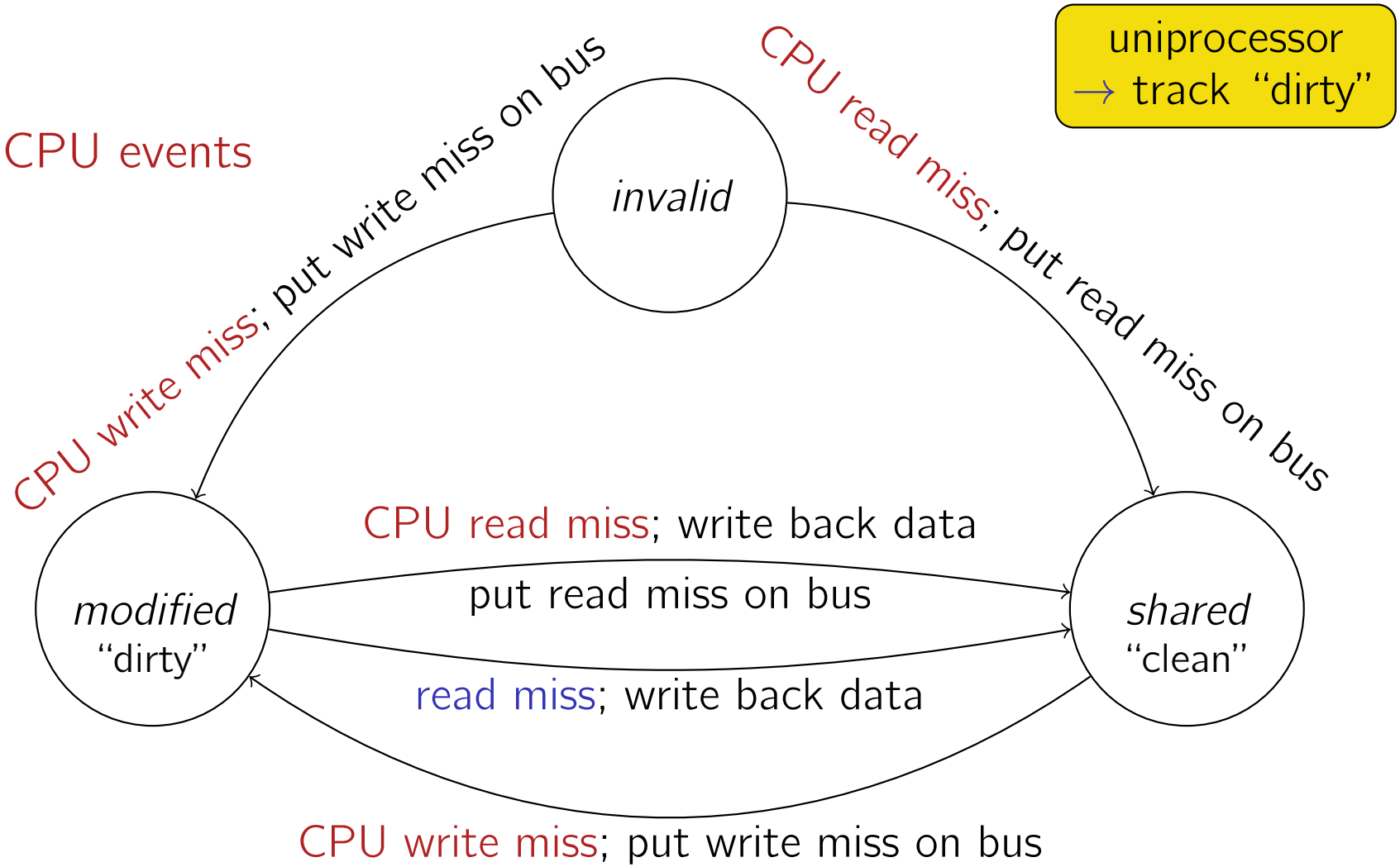
→ Track **sharing state**.

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*.

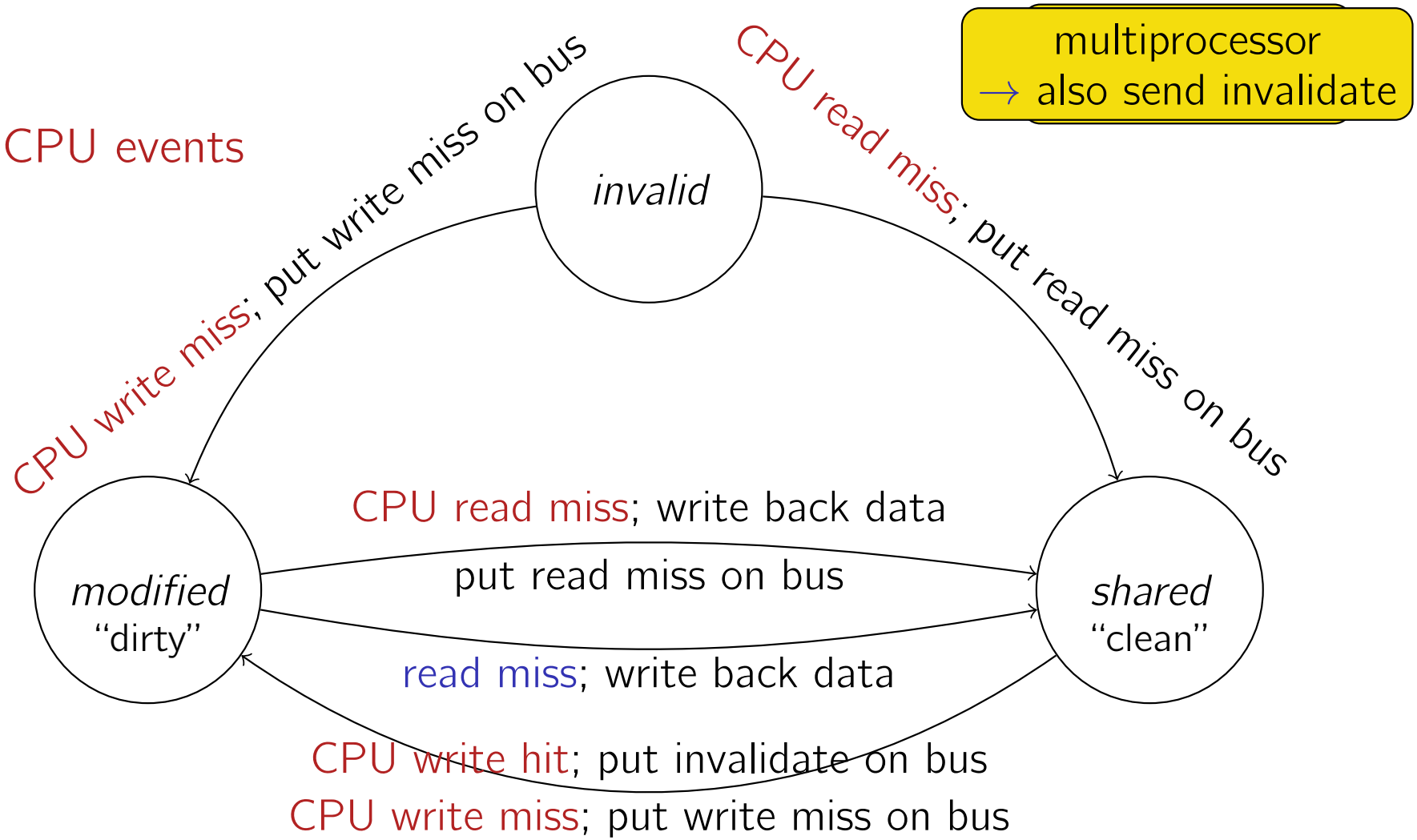
Write Invalidate—State Machine

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



Write Invalidate—State Machine

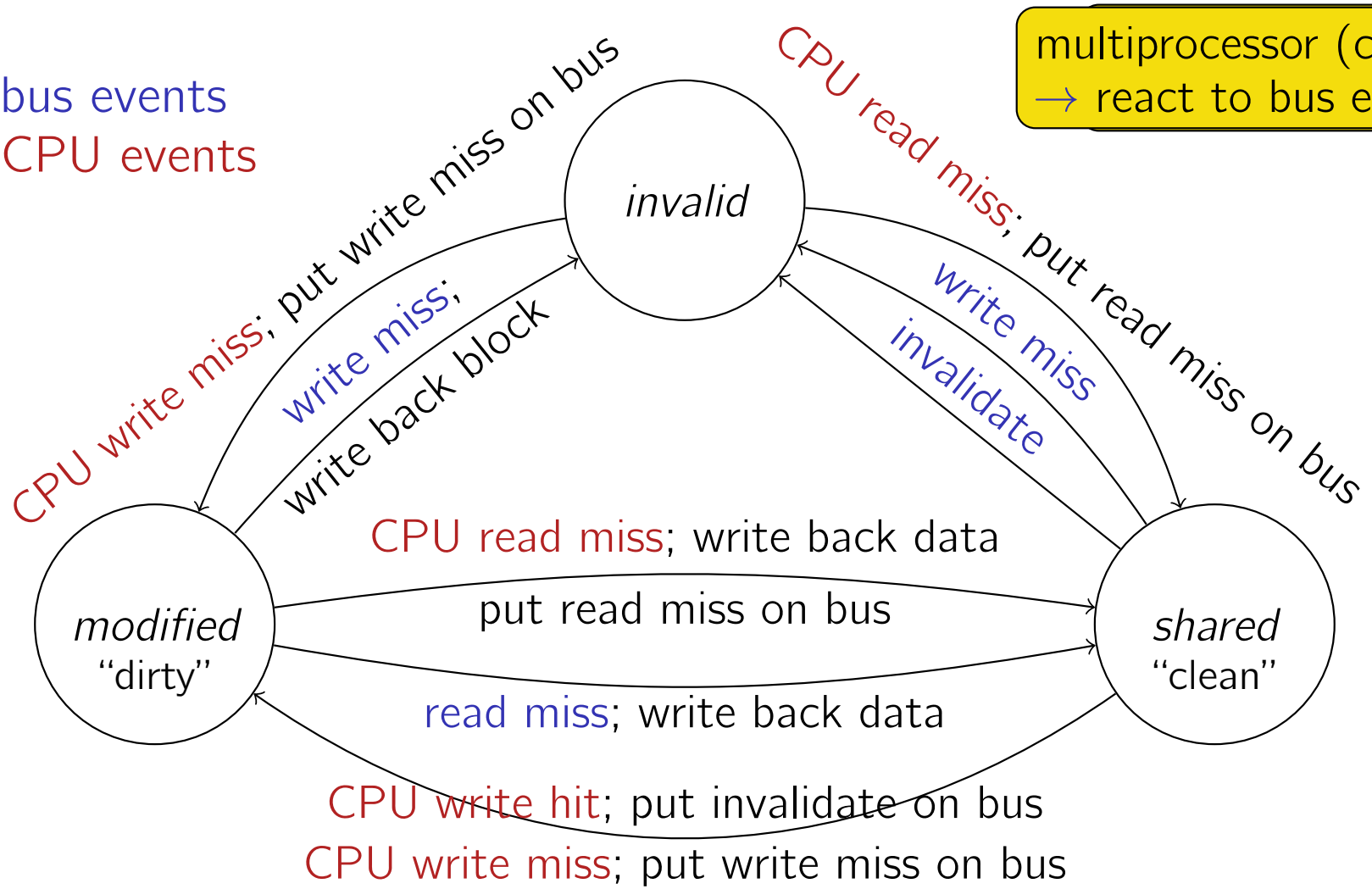
Local caches track sharing states using a **state machine**.



Write Invalidate—State Machine

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

multiprocessor (cont.)
→ react to bus events



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 ($\rightsquigarrow \lhd$).

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!

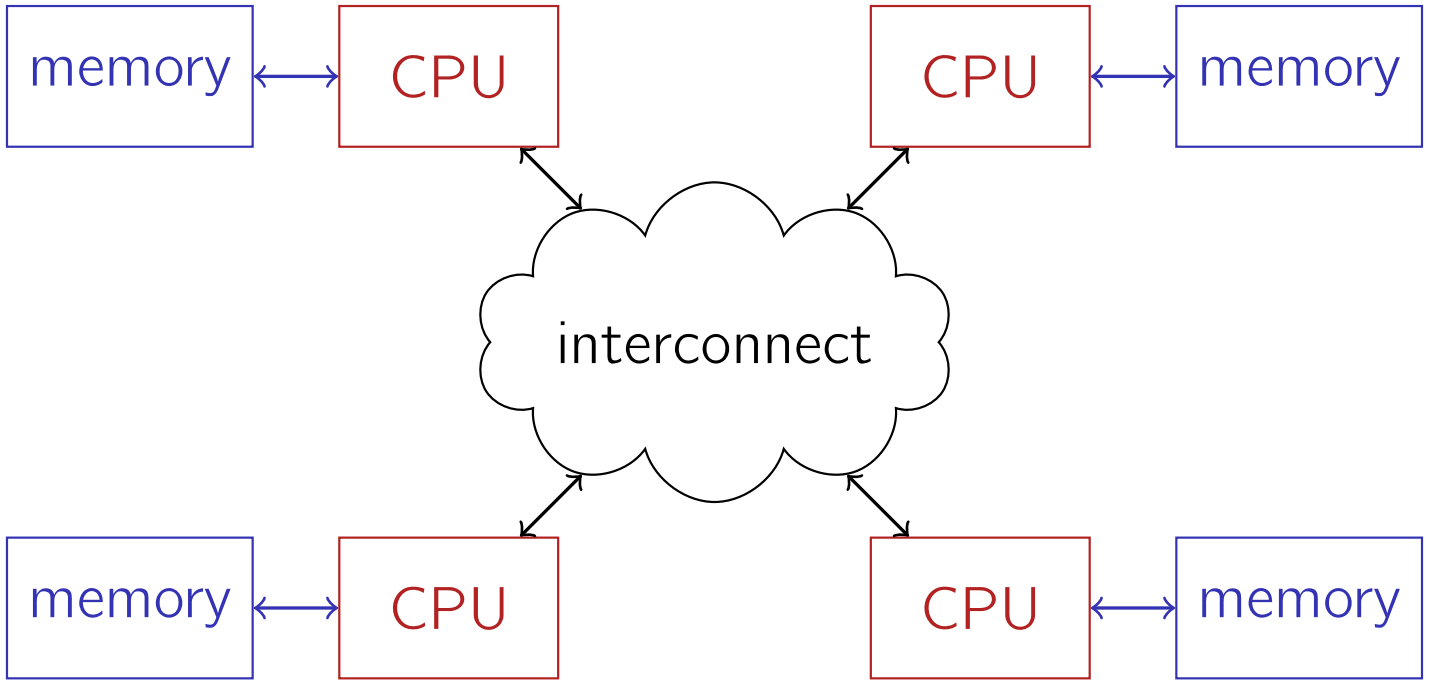
Shared memory bus?

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

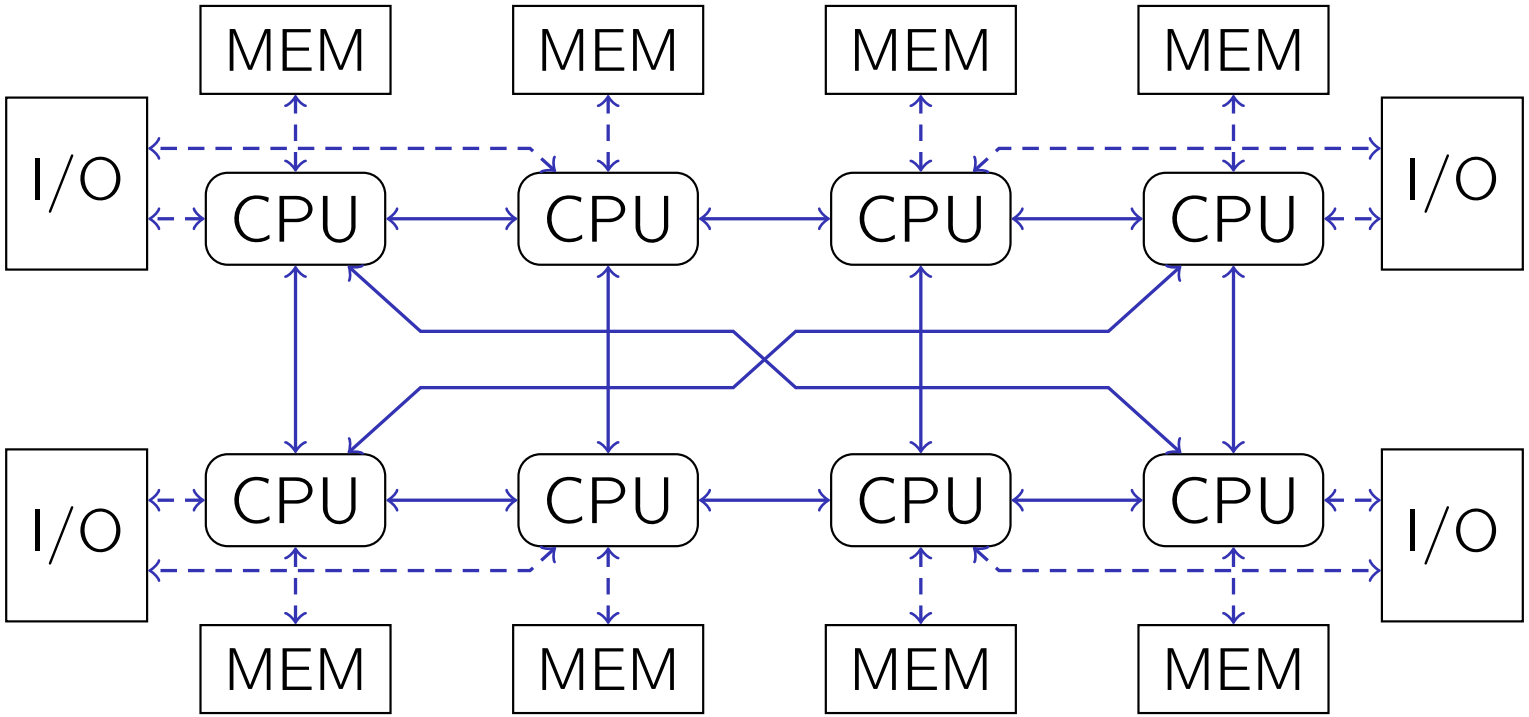
Distributed Shared Memory

Idea: Distribute memory

→ Attach to individual compute nodes



Example: 8-Way Intel Nehalem-EX



- Interconnect: "Intel Quick Path Interconnect (QPI)" [9]
- Memory may be local, one hop away, or two hops away.
  → Non-uniform memory access (NUMA)

[9] The AMD counterpart is "HyperTransport".

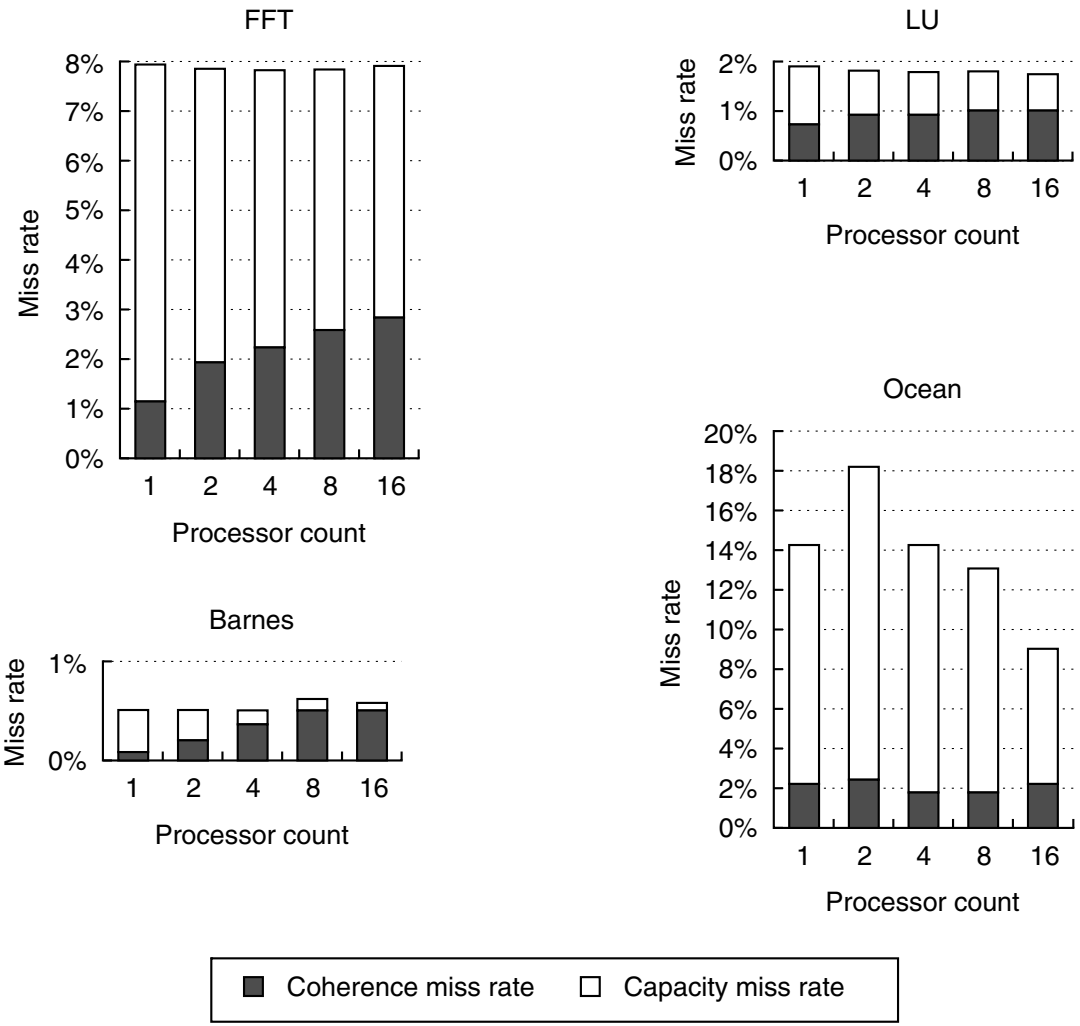
Distributed Memory and Snooping

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** ($\leadsto$ 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

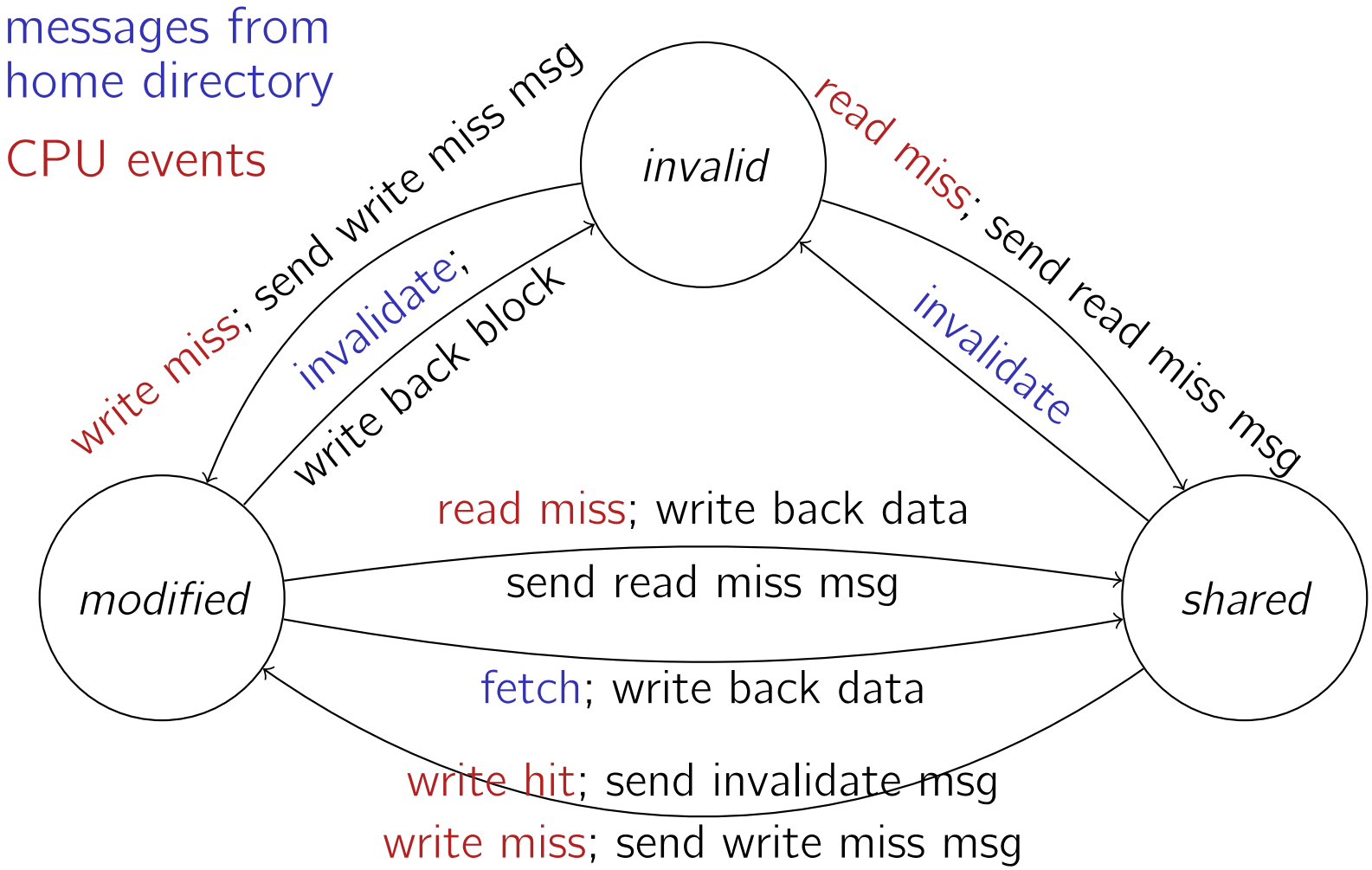
Messages sent by individual nodes:

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.

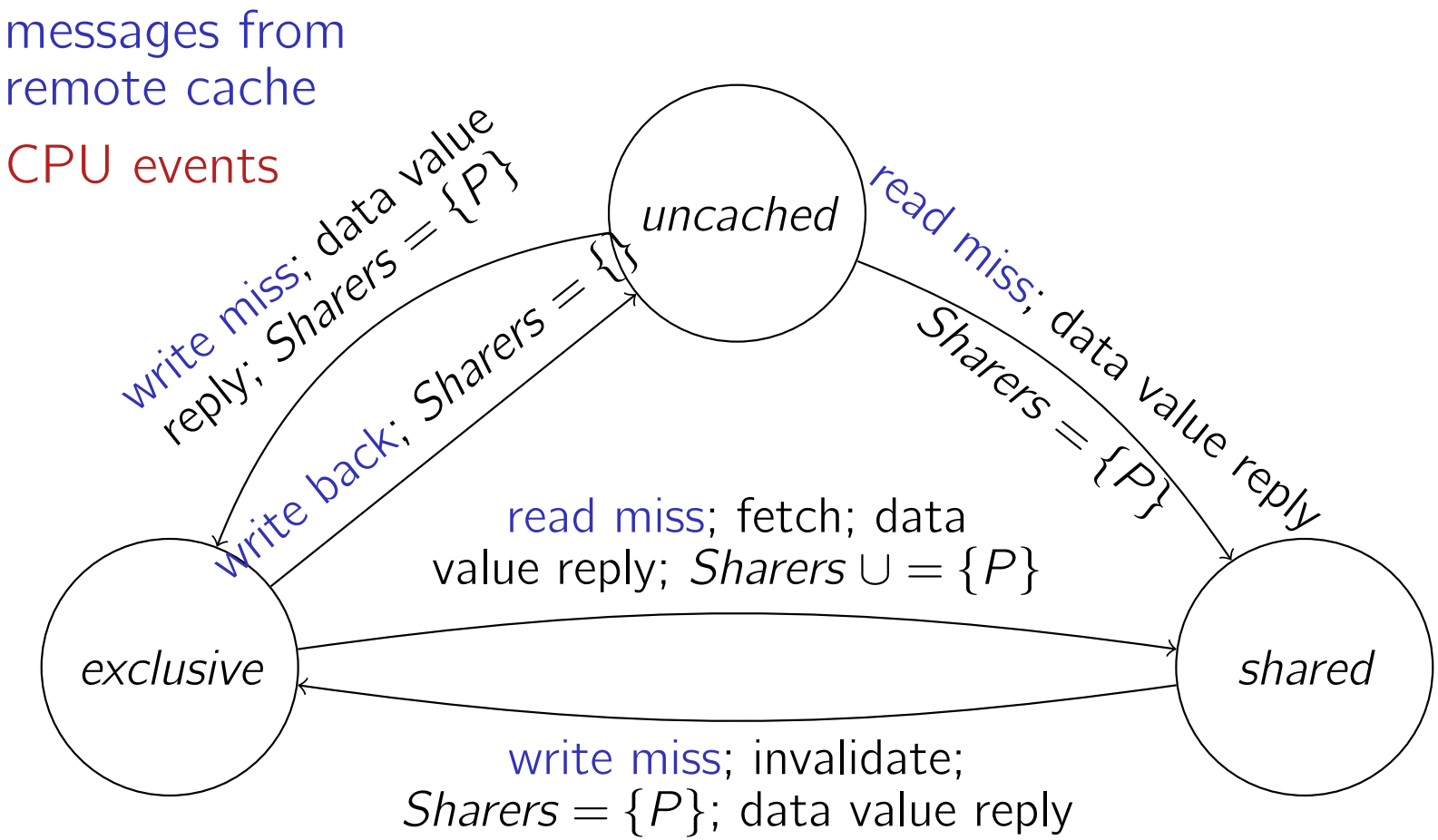
Directory-Based Coherence—State Machine

Individual caches use a state machine similar to the one on slide 213.



Directory-Based Coherence—State Machine

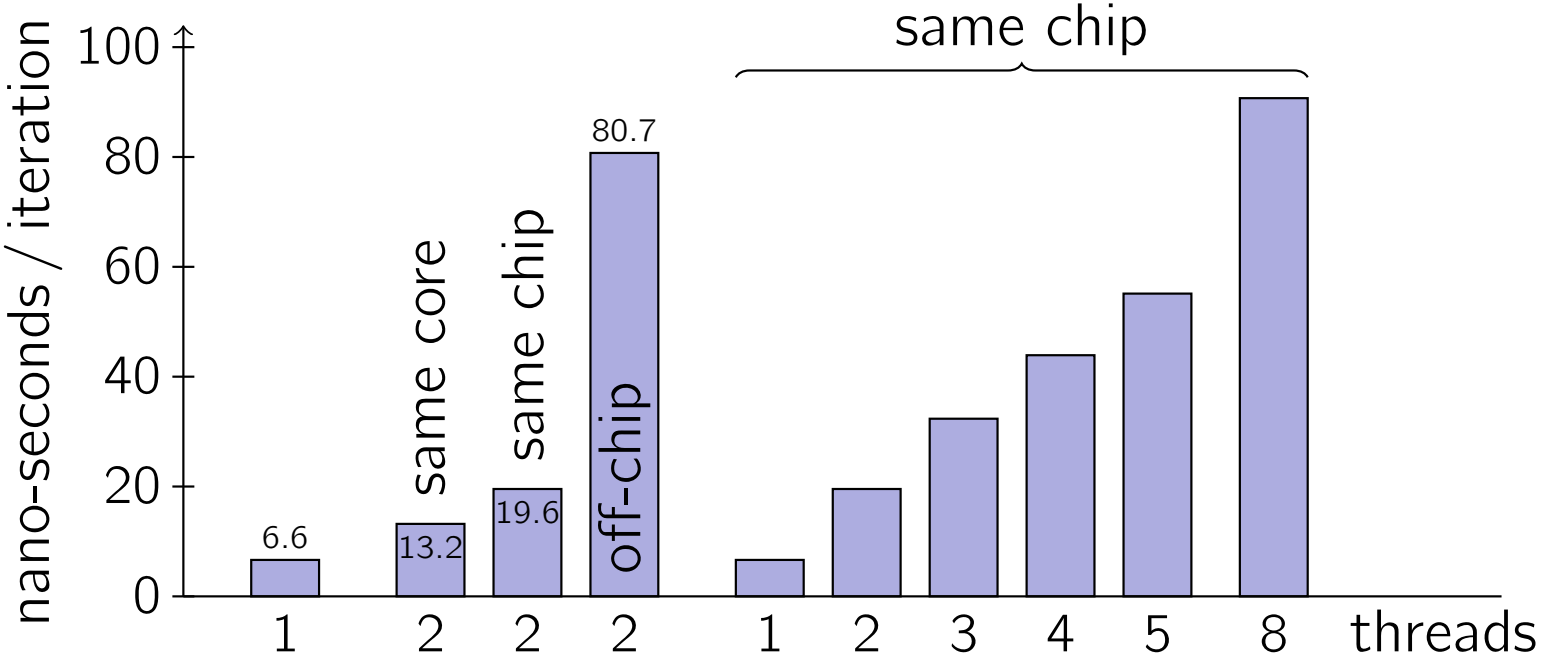
The **directory** has its own state machine.



Cache Coherence Cost

Experiment:

- Several threads randomly increment elements of an integer array; Zipfian probability distribution, no synchronization[10].



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

[10] 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**.

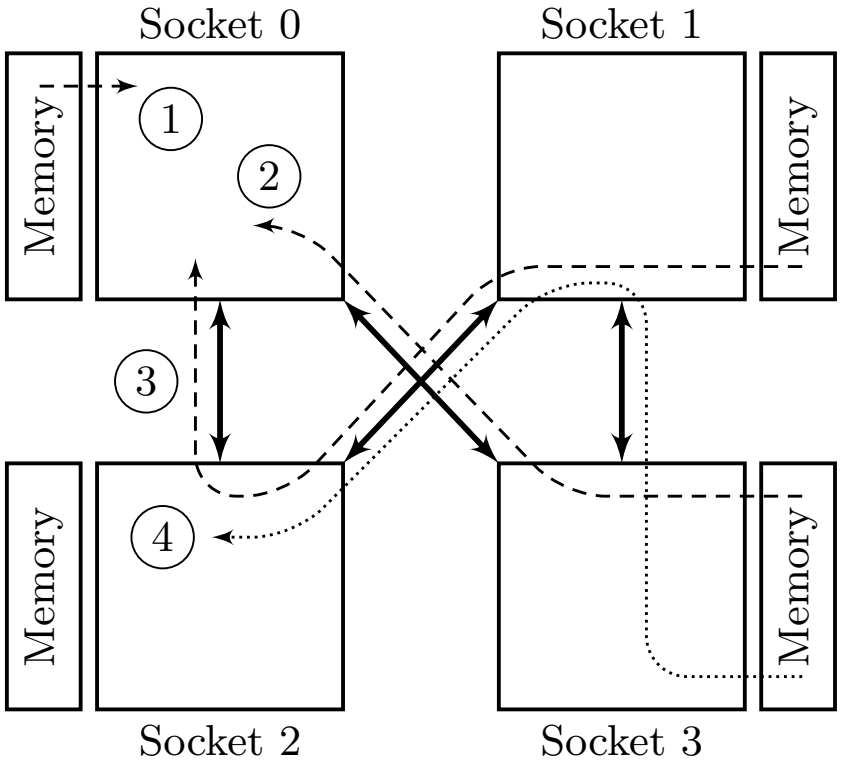
→ Items get invalidated/migrated, even though no data is actually shared.

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

NUMA—Non-Uniform Memory Access

Distribution makes memory access **locality-sensitive**.

→ **Non-Uniform Memory Access (NUMA)**

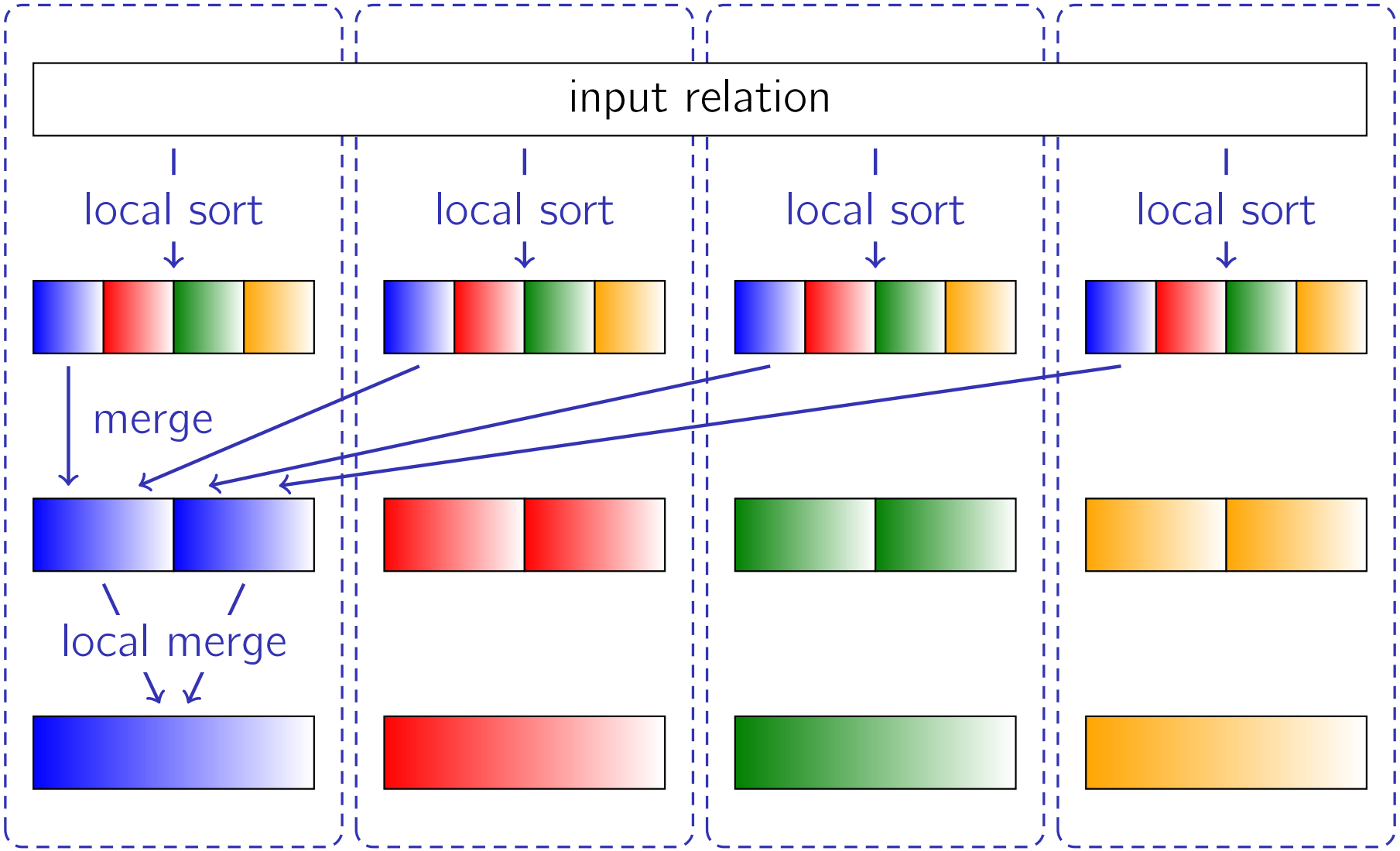


	bandwidth	latency
①	24.7 GB/s	150 ns
②	10.9 GB/s	185 ns
③	10.9 GB/s	230 ns
③/④[11]	5.3 GB/s	235 ns

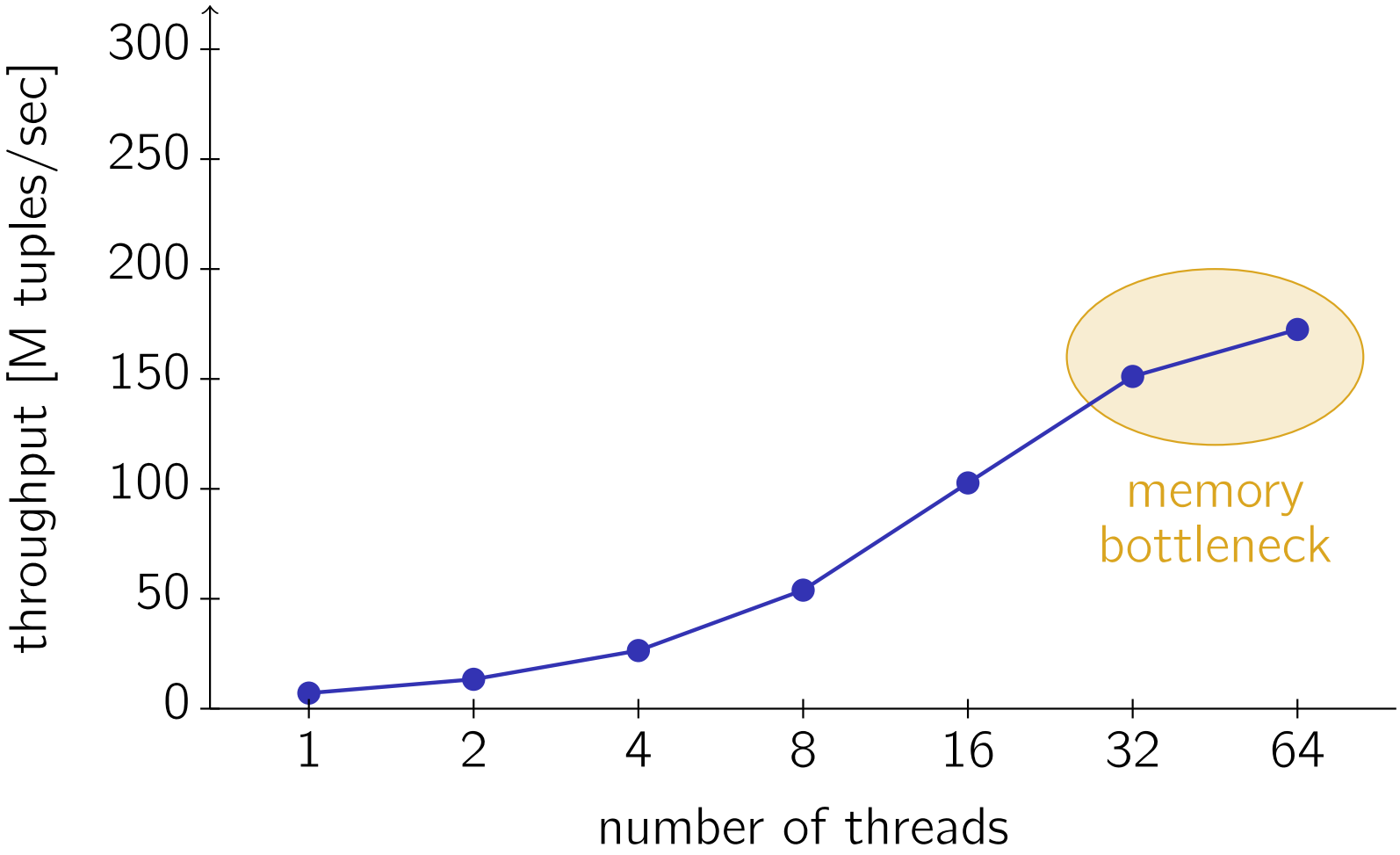
↗ Li *et al.* NUMA-Aware Algorithms: The Case of Data Shuffling. *CIDR 2013*

[11] ③ with cross traffic along ④.

Sorting and NUMA



Resulting Throughput

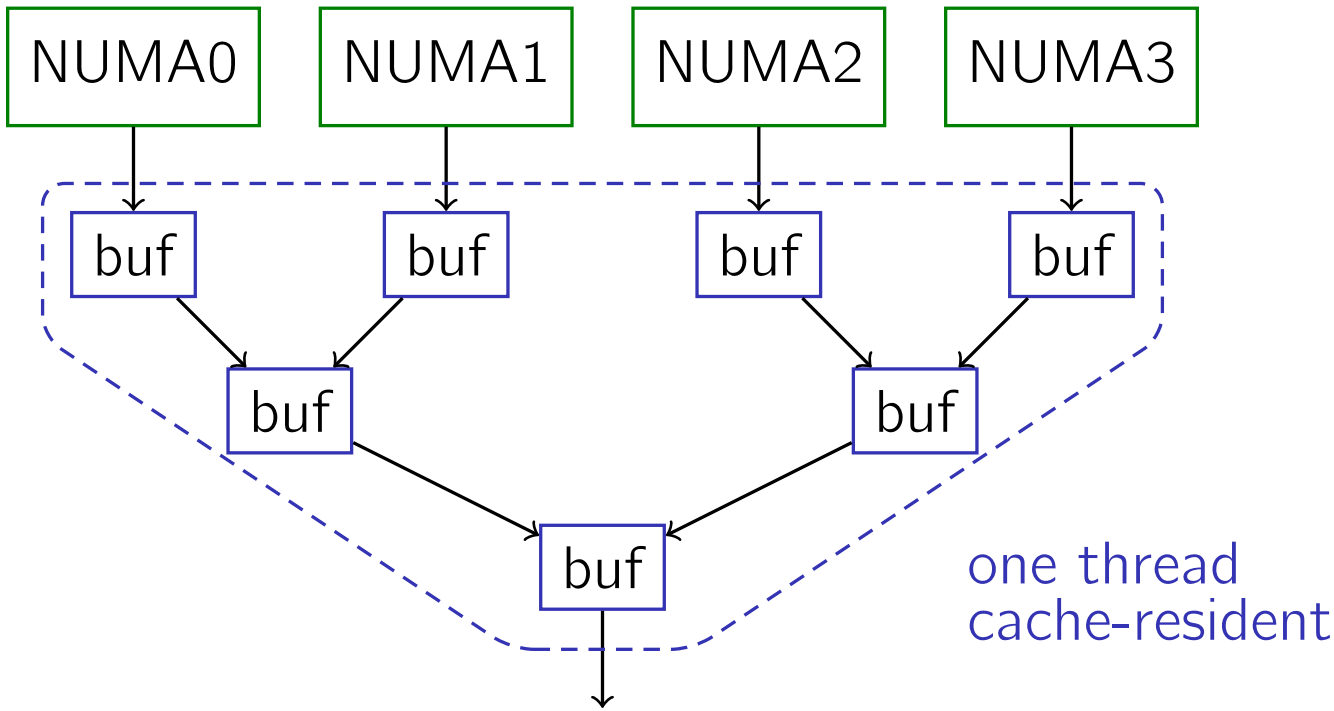


NUMA and Bandwidth

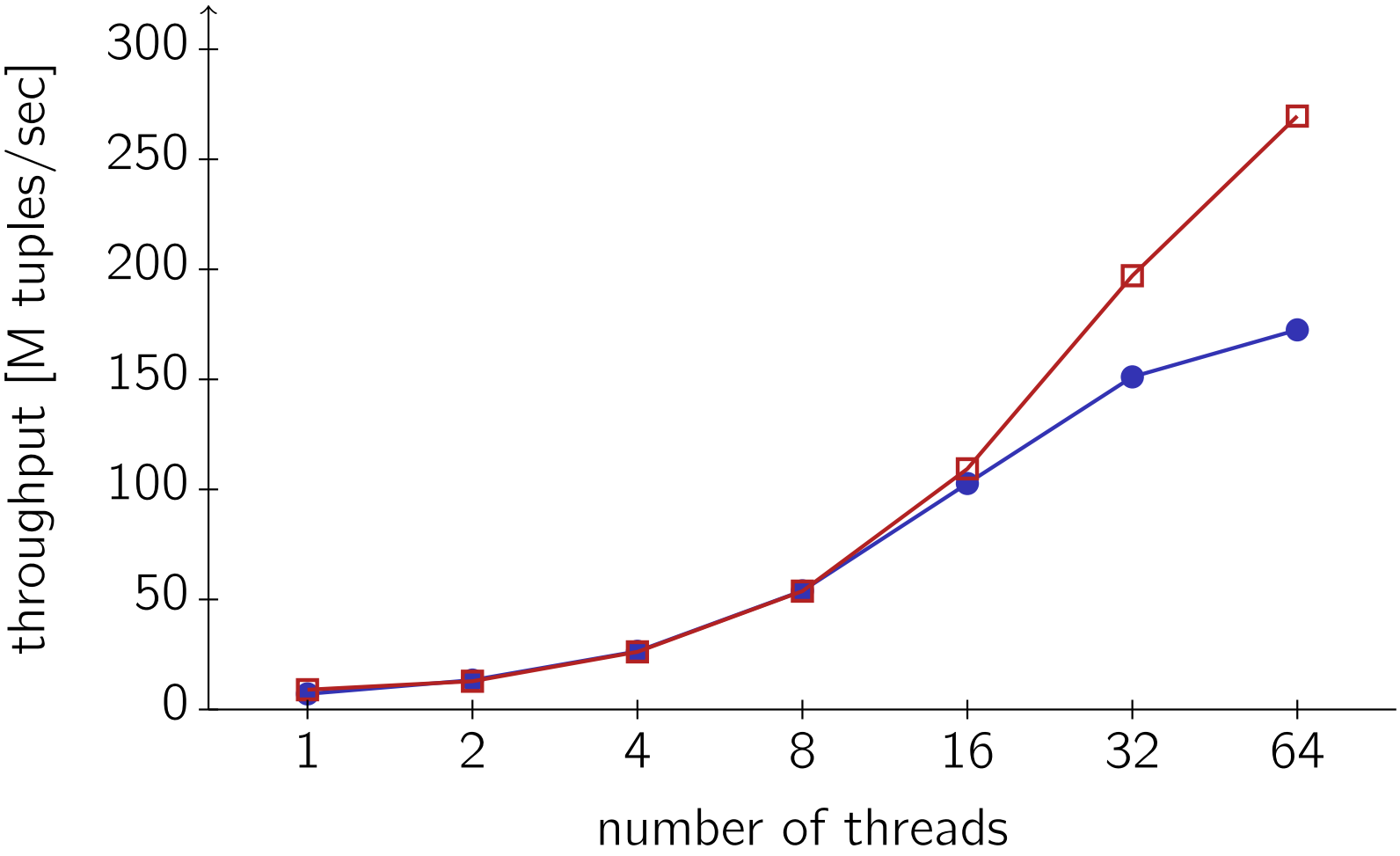
Problem: Merging is **bandwidth-bound**.

→ Merge multiple runs (from NUMA regions) at once
(Two-way merging would be more CPU-efficient because of SIMD.)

→ Might need **more instructions**, but brings bandwidth and compute **into balance**.



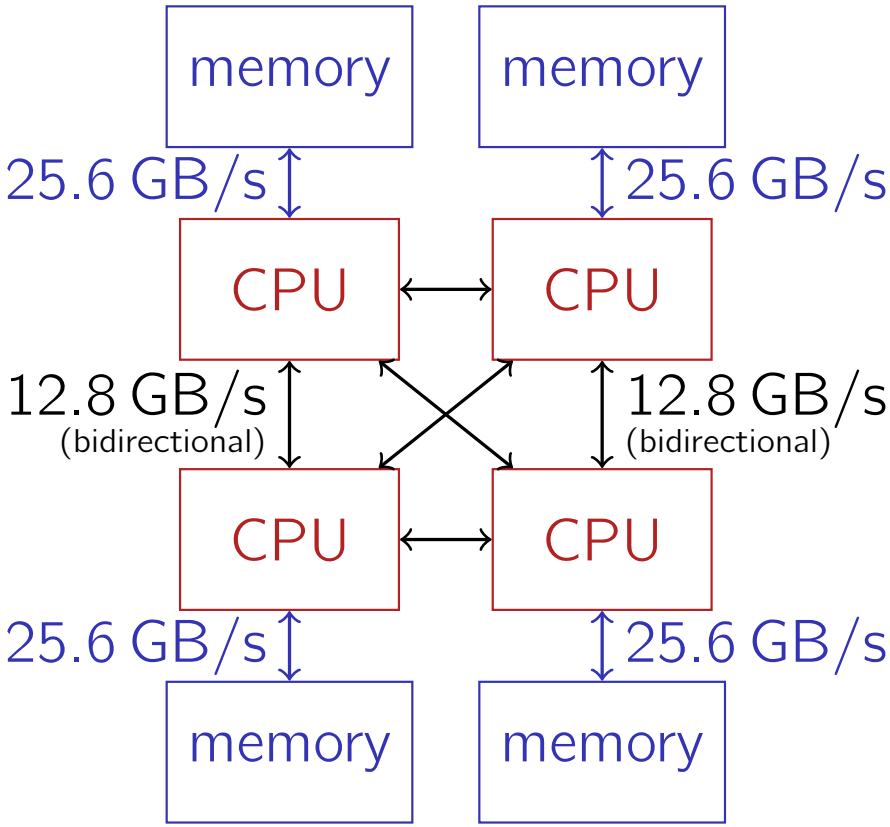
Throughput With Multi-Way Merging



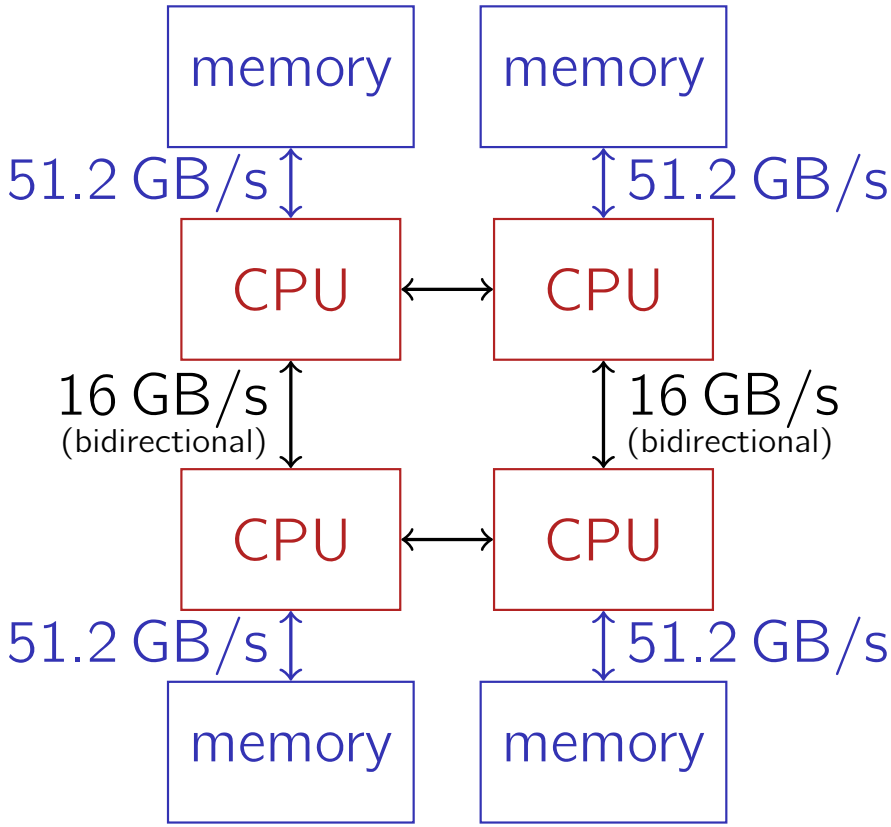
NUMA Effects in Detail

Bandwidth:

- Single links have **lower bandwidth** than memory controllers.



Intel Nehalem EX



Intel Sandy Bridge EP

Parallelize Database Workloads

To leverage the hardware potential, databases **must** use parallelism.

→ **Inter-Query Parallelism?**

- Requires a sufficient number of **co-running queries**.
- May work well for **OLTP workloads**
  (They tend to be characterized by many, many queries, each of which is very simple.)
- Data Analytics/OLAP often don't fulfill the requirement.
- Won't help an individual query.

Therefore:

- **Intra-query parallelism** is a **must**.
- Should still allow (few) **co-running queries**.

Parallelization Strategies

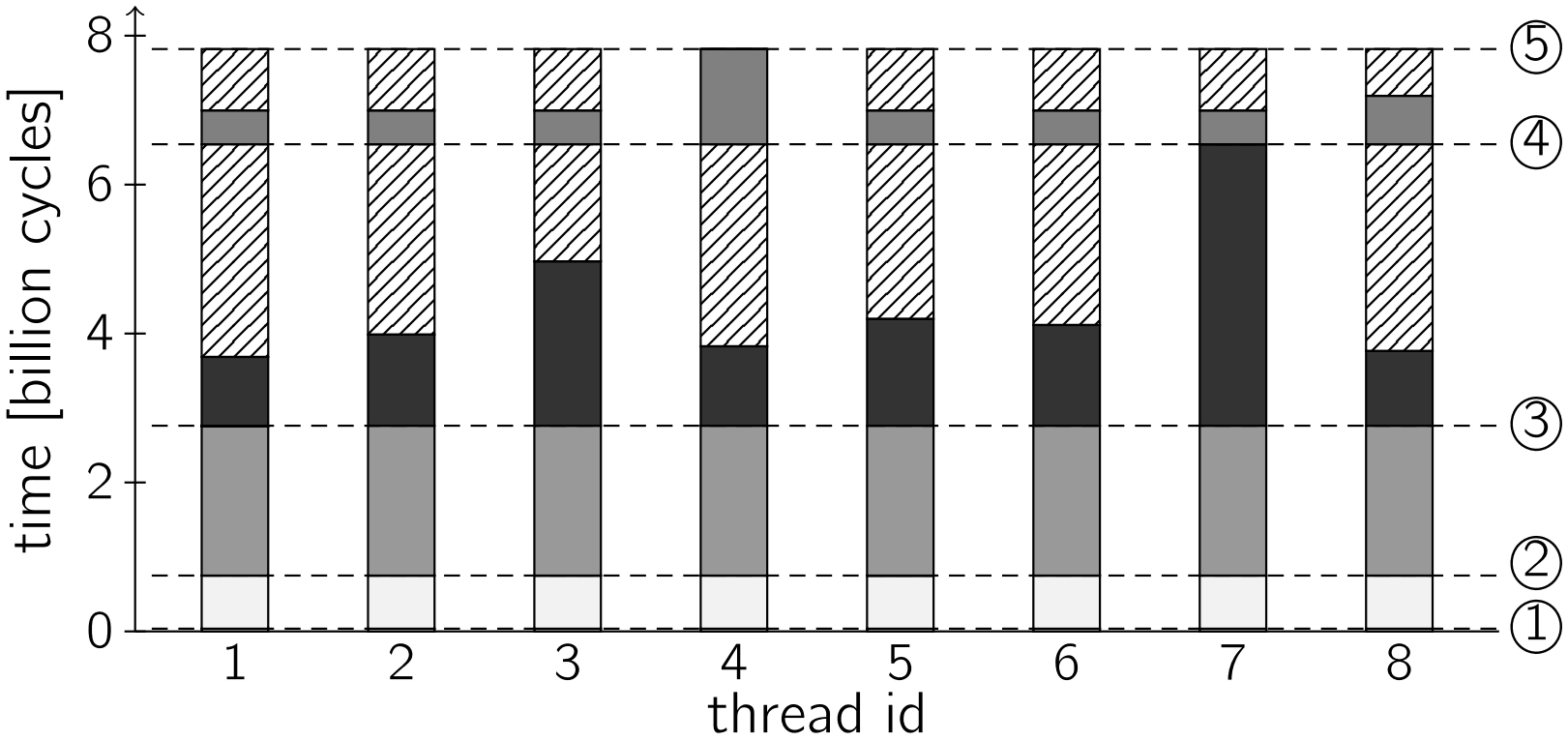
Parallelization strategies for **intra-query parallelism**:

- **Pipeline Parallelism?** 

- **Data Partitioning / Parallel Operator Implementations?** 

Data Partitioning / Parallel Operator Implementations

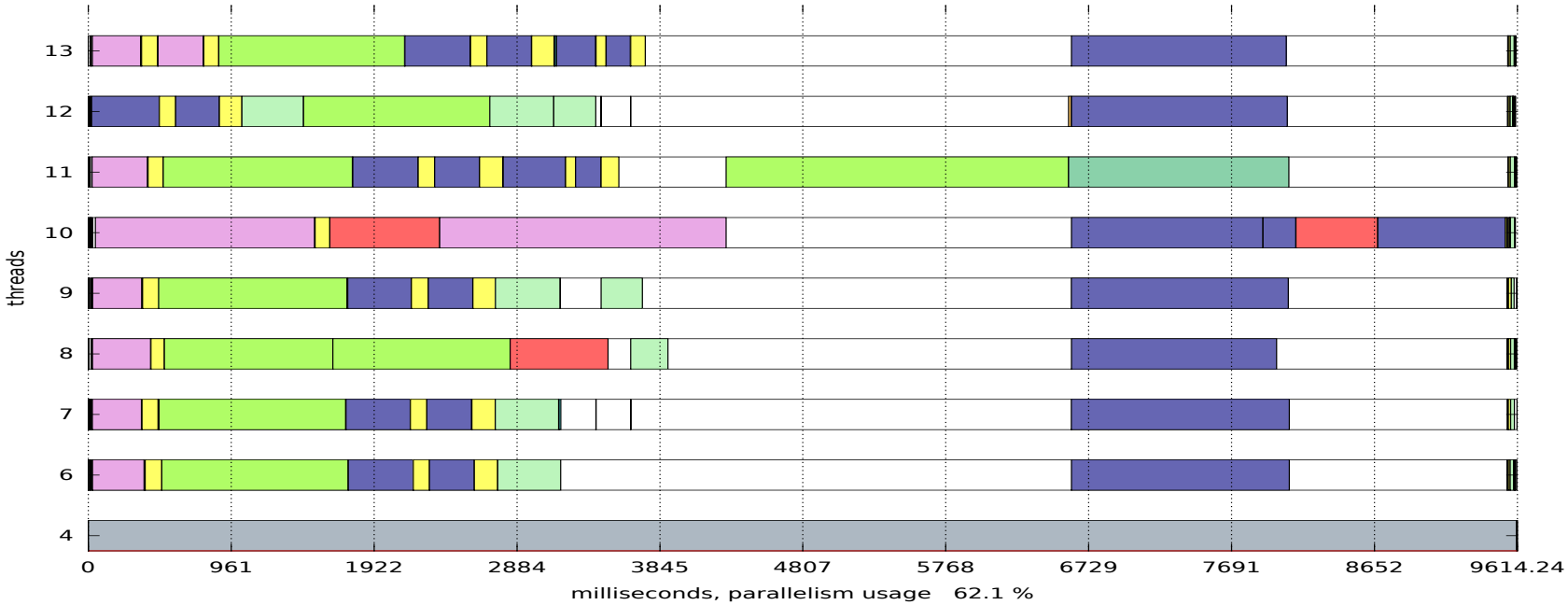
E.g., parallel hash joins (radix joins):



↗ Balkesen *et al.* Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. *ICDE 2013*.

Data Partitioning / Parallel Operator Implementations

E.g., TPC-H Query 10 on MonetDB:



↗ Mrunal Gawade. Multi-Core Parallelism in a Column-Store. *PhD Thesis*, Universiteit van Amsterdam. 2017.

Data Partitioning

Lessons learned:

→ Use **fine-granular** partitioning.

- Increased scheduling overhead seems bearable.

→ Assign partitions/tasks **dynamically** to processors.

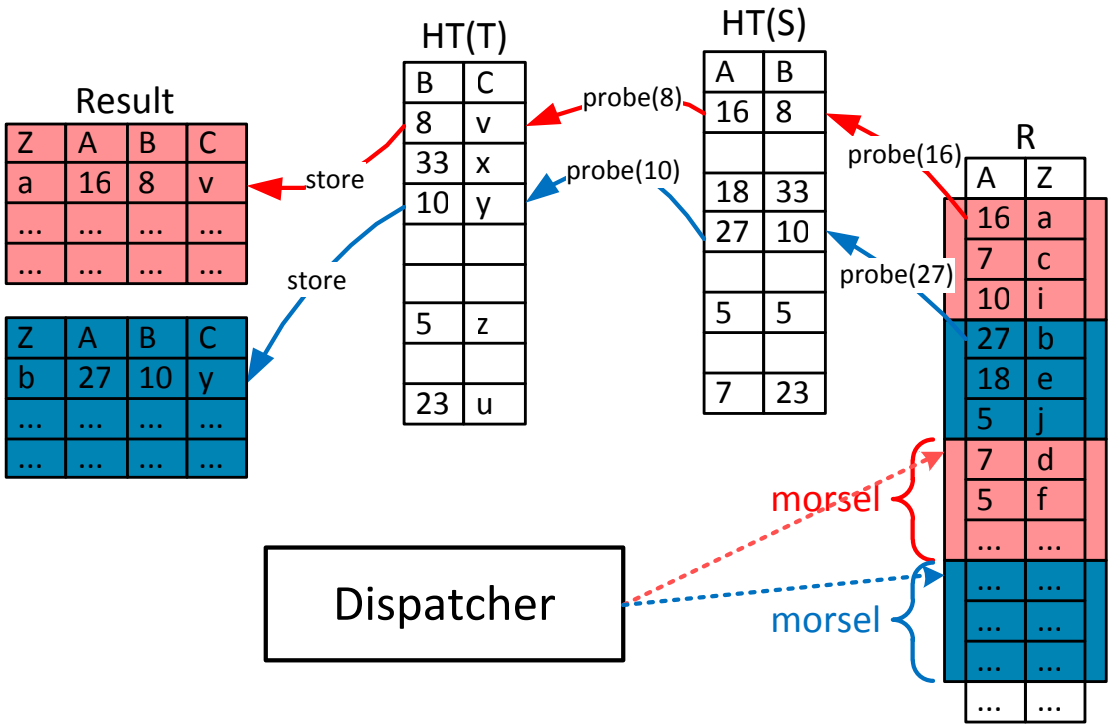
- Makes load balancing easier.

E.g., **Morsel-Driven Parallelism:** (as implemented in HyPer)

- Break operator inputs into chunks of ≈100,000 tuples ("morsels").
- Fixed number of operator threads.
- Morsels dispatched to threads dynamically (task queue).

↗ Leis *et al.* Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. *SIGMOD 2014*.

Morsel-Driven Parallelism: Idea



- Probe phase of join query $R \bowtie S \bowtie T$.
- R broken up in segments; threads grab segments and, for each tuple, probe into hash tables $HT(S)$ and $HT(T)$.

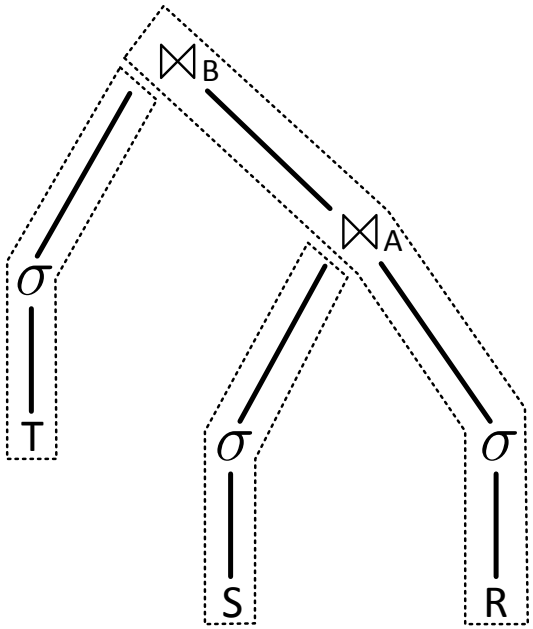
Morsel-Driven Parallelism: Query Pipelines

HyPer compiles plan segments between **pipeline breakers** into machine code.

E.g., three pipelines:

1. Scan, filter T, build $HT(T)$.
2. Scan, filter S, build $HT(S)$.
3. Scan, filter R, probe into both hash tables.

After compilation, each pipeline becomes one "operator".



Data dependencies:

→ Pipelines 1 and 2 must complete before Pipeline 3 begins.

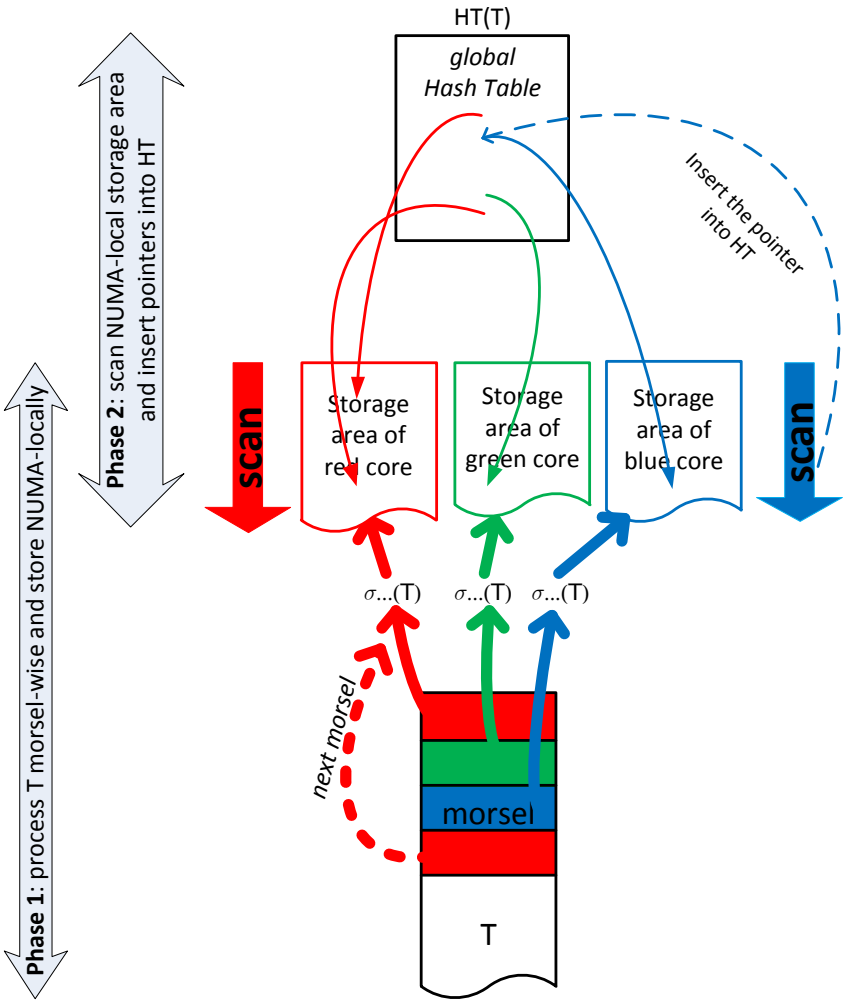
→ But Pipelines 1 and 2 can run in parallel.

Avoid Synchronization / Increase Locality

HyPer, in fact, breaks up hash table builds into two phases:

- Threads implement σ first and move tuples to **private storage area**.
- Build **global hash table** afterward.

✎ **Advantages?**



Morsel-Driven Parallelism: NUMA Awareness

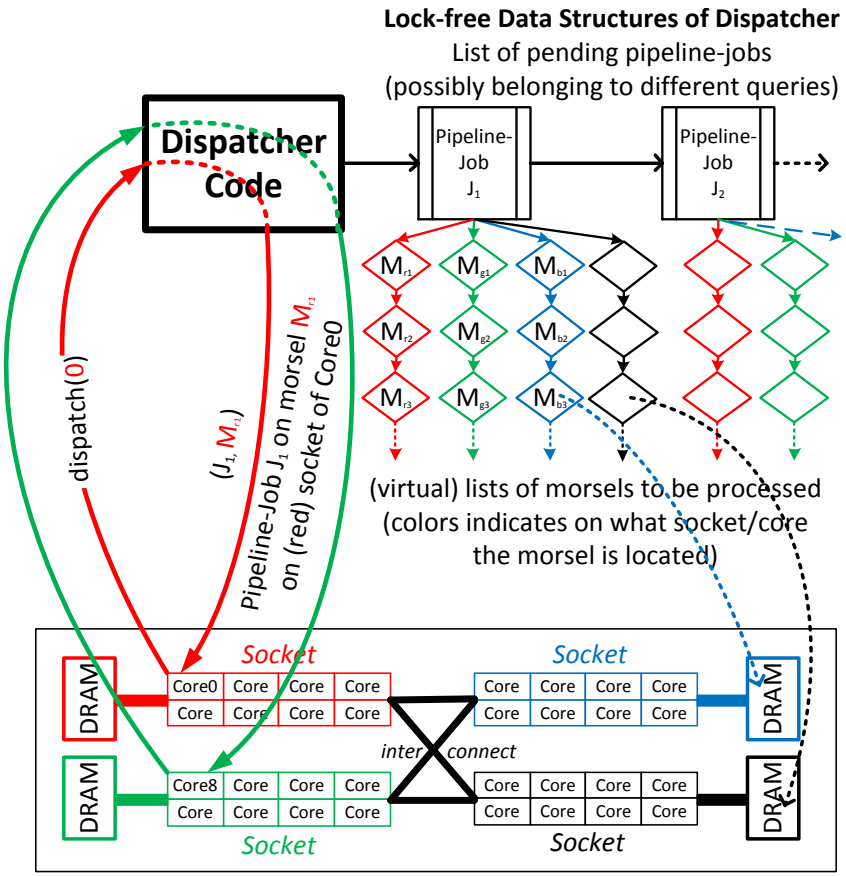
NUMA Awareness:

- Annotate morsels with **NUMA region** where their data resides.
- When dispatching to cores favor NUMA-local assignments.

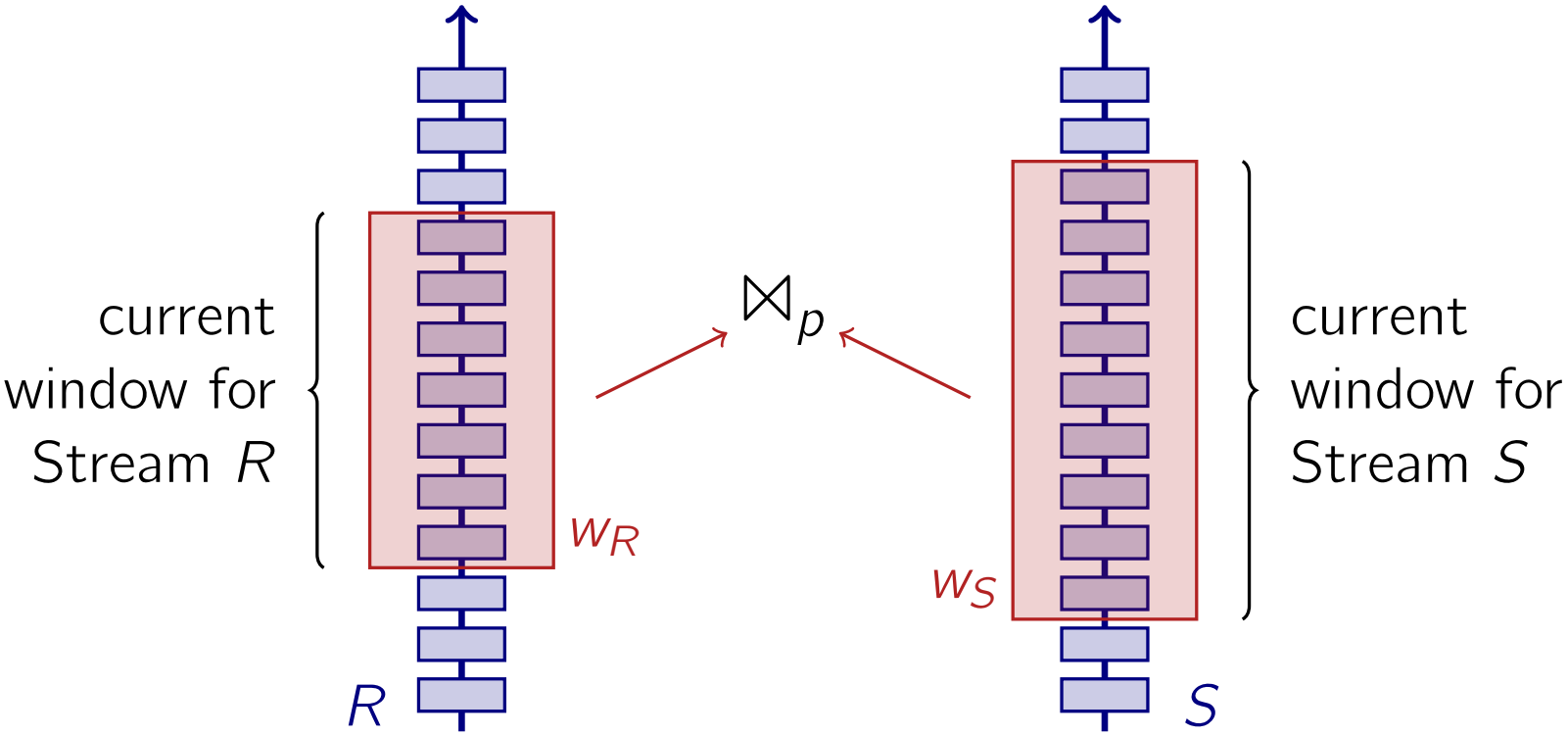
Elasticity:

- There's only **one task pool** for the entire system.
- Multiple queries share the same worker threads.

→ Parallelism across and within queries.

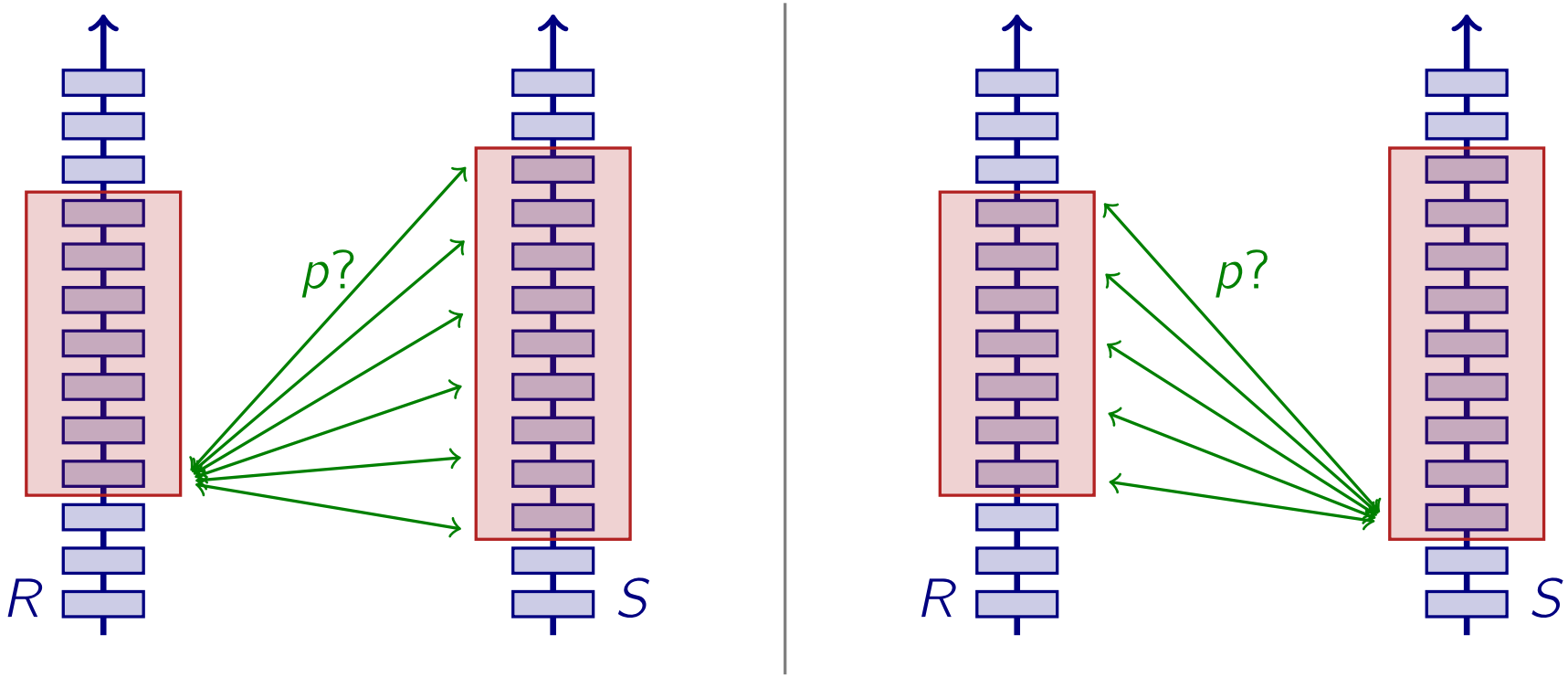


Joins Over Data Streams:



Task: Find all $\langle r, s \rangle$ in w_R, w_S that satisfy $p(r, s)$.

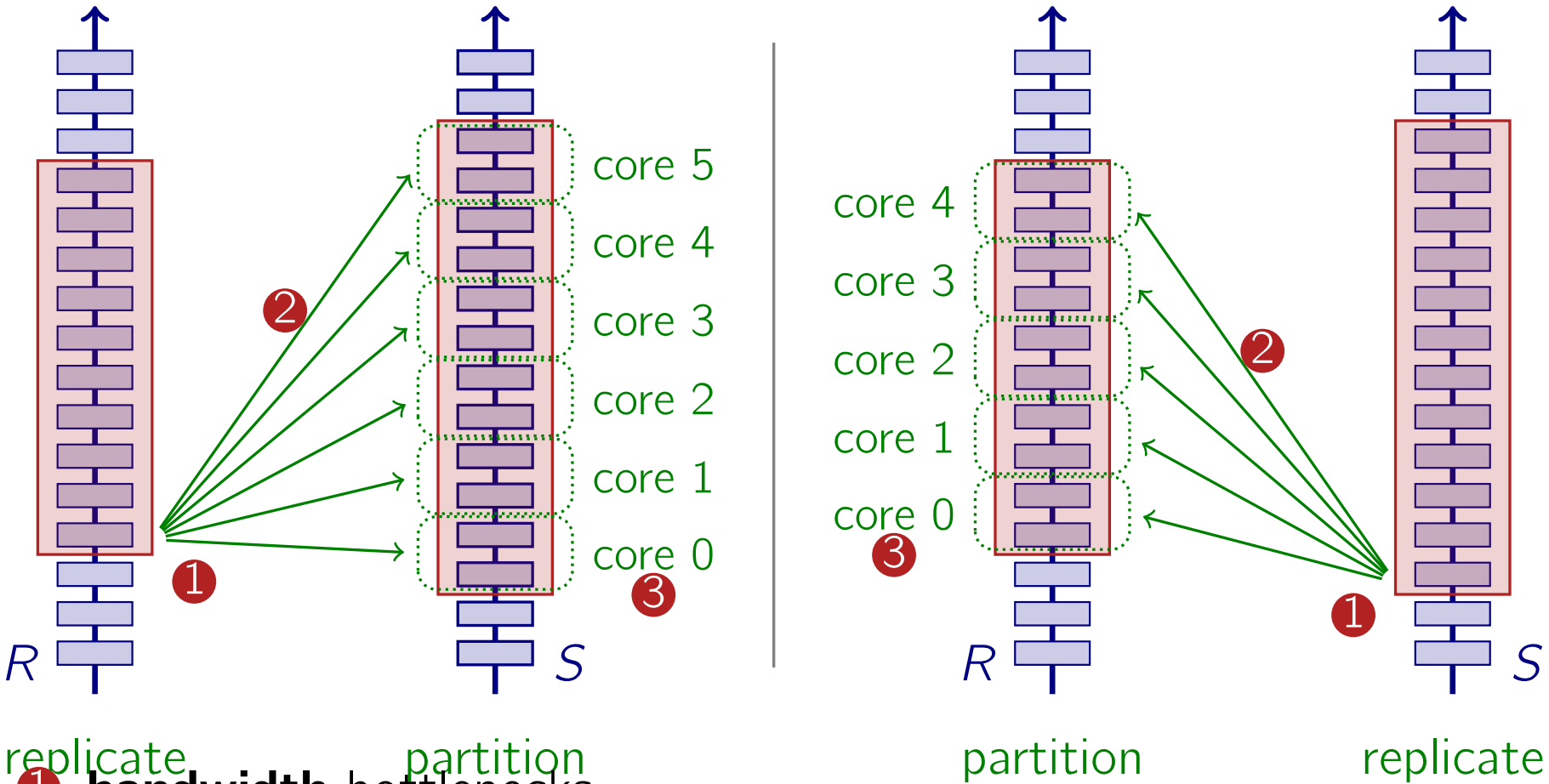
Implementation [Kang *et al.*, ICDE 2003]



1. scan window, **2. insert** new tuple, **3. invalidate** old

NUMA-Aware Execution?

CellJoin [Gedik *et al.*, VLDBJ 2009]

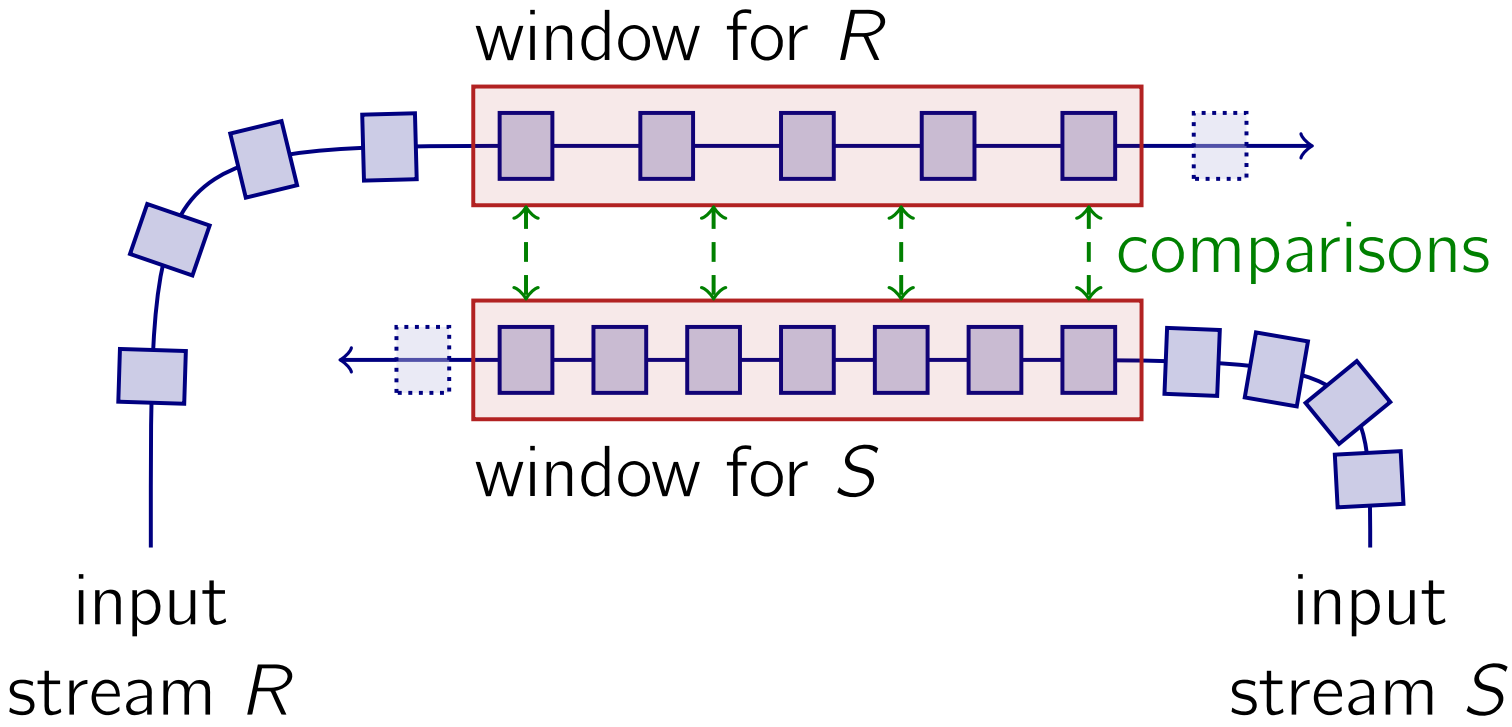


1. **bandwidth** bottlenecks
2. **long-distance** communication
3. **centralized** coordination and memory

→ Parallel, but not NUMA-aware.

Handshake Join Idea

Handshake Join:

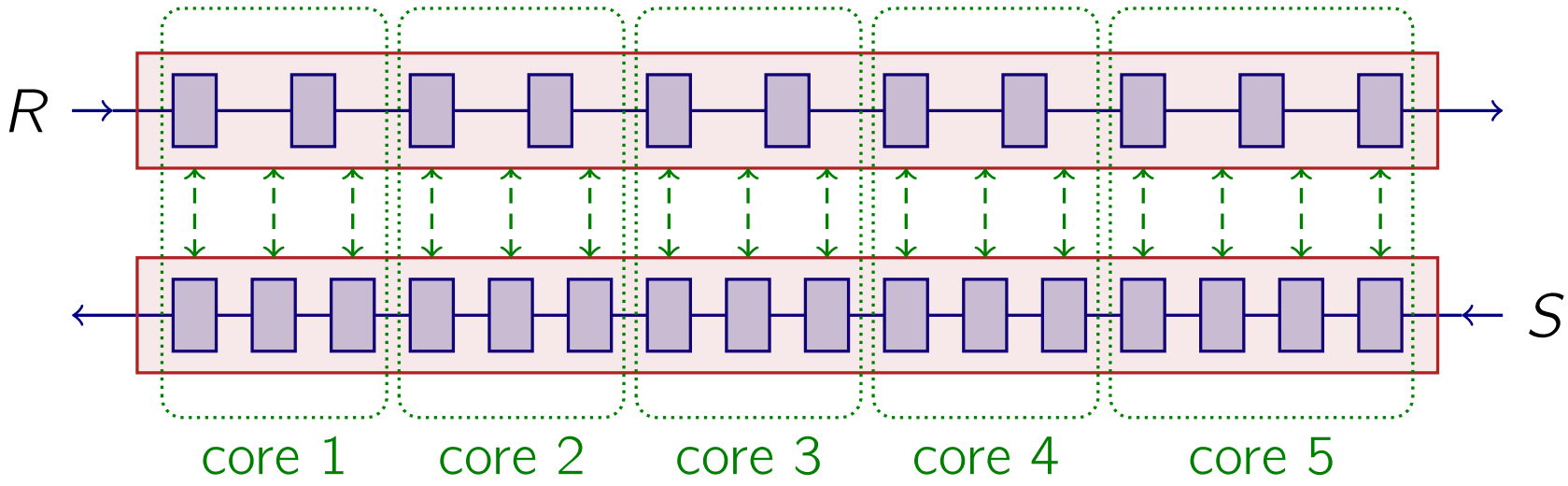


Streams flow by in **opposite directions**

Compare tuples when they **meet**

Handshake Join on Many Cores

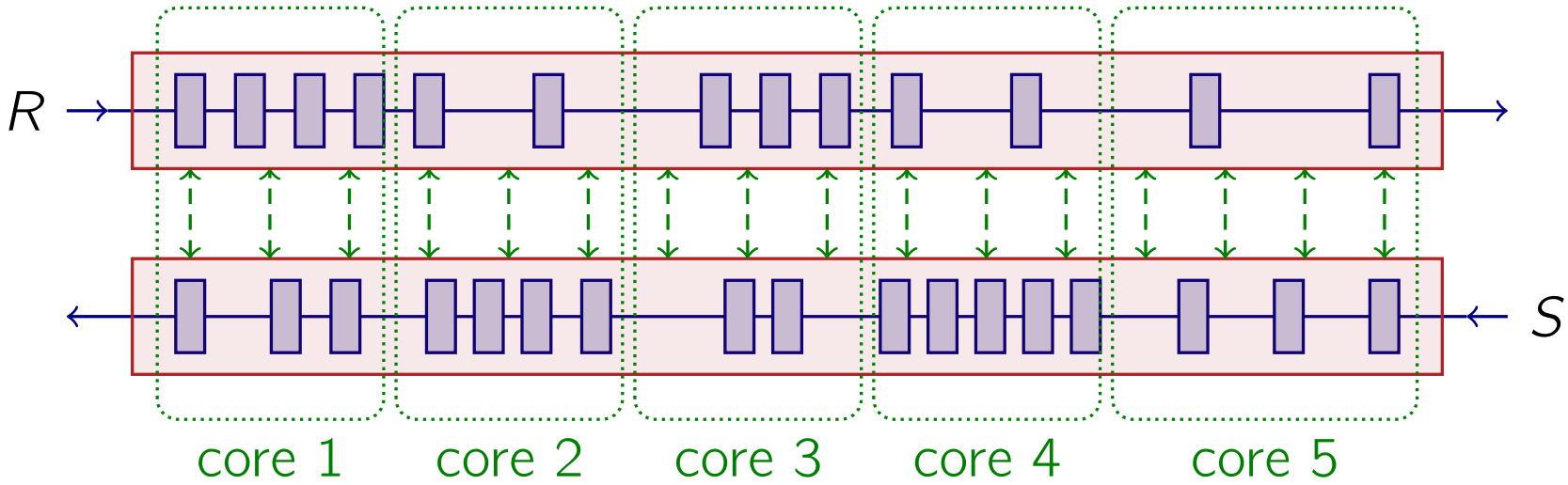
Data flow representation → **parallelization**:



- **No bandwidth bottleneck** ❶ ✓
- Communication/synchronization stays **local** ❷ ✓

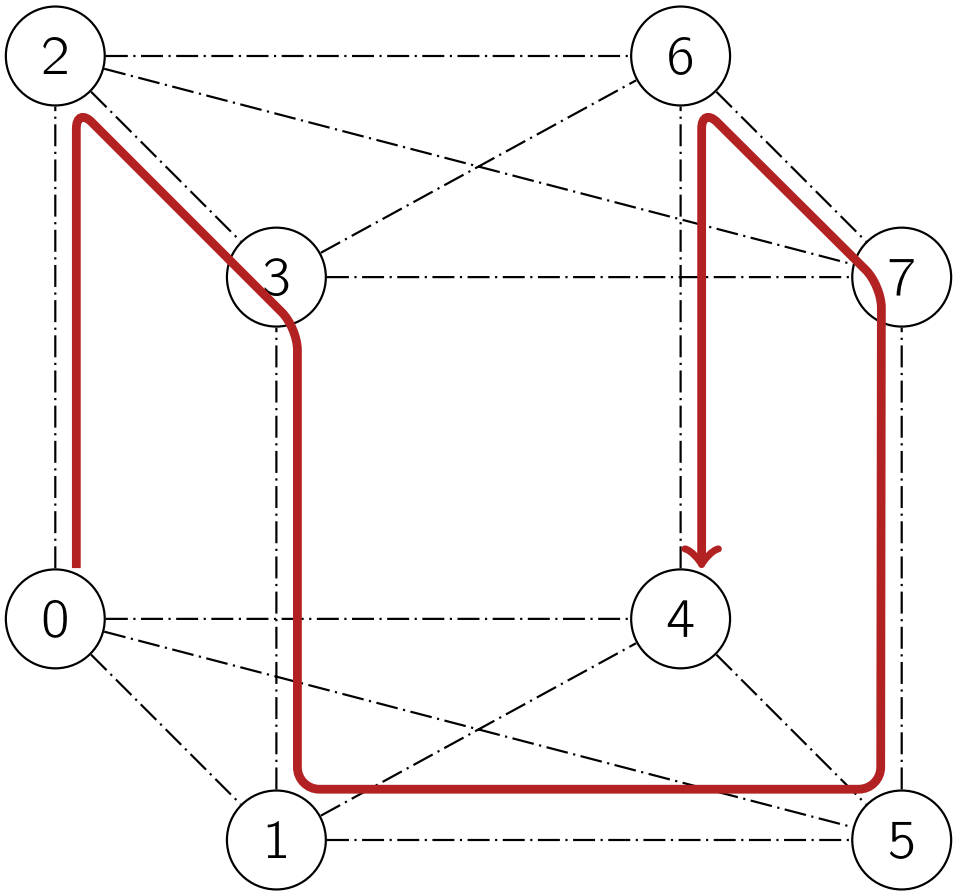
Synchronization

Coordination can now be done **autonomously**

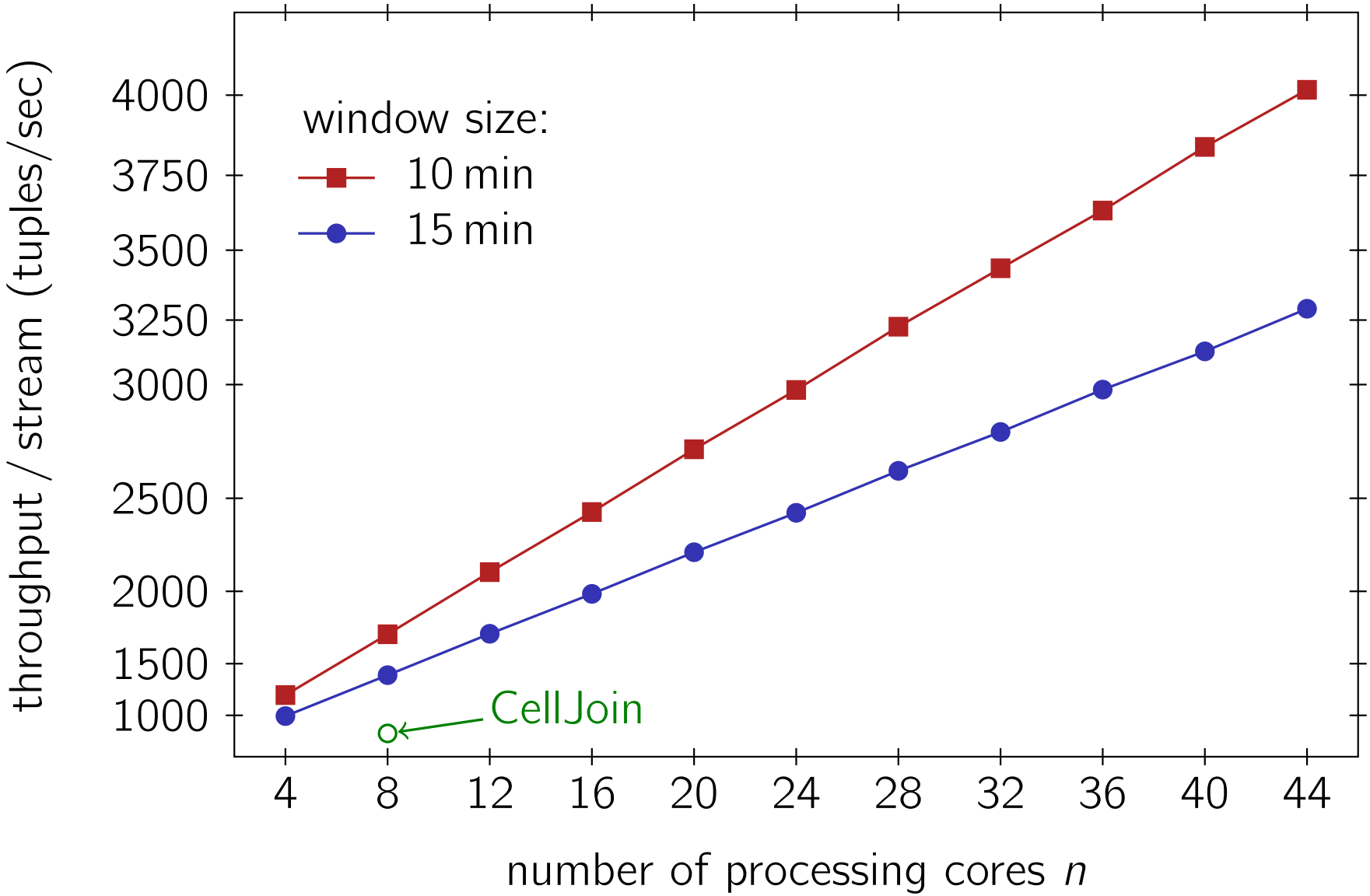


- no more centralized coordination ❸ ✓
- Autonomous **load balancing**
- **Lock-free message queues** between neighbors

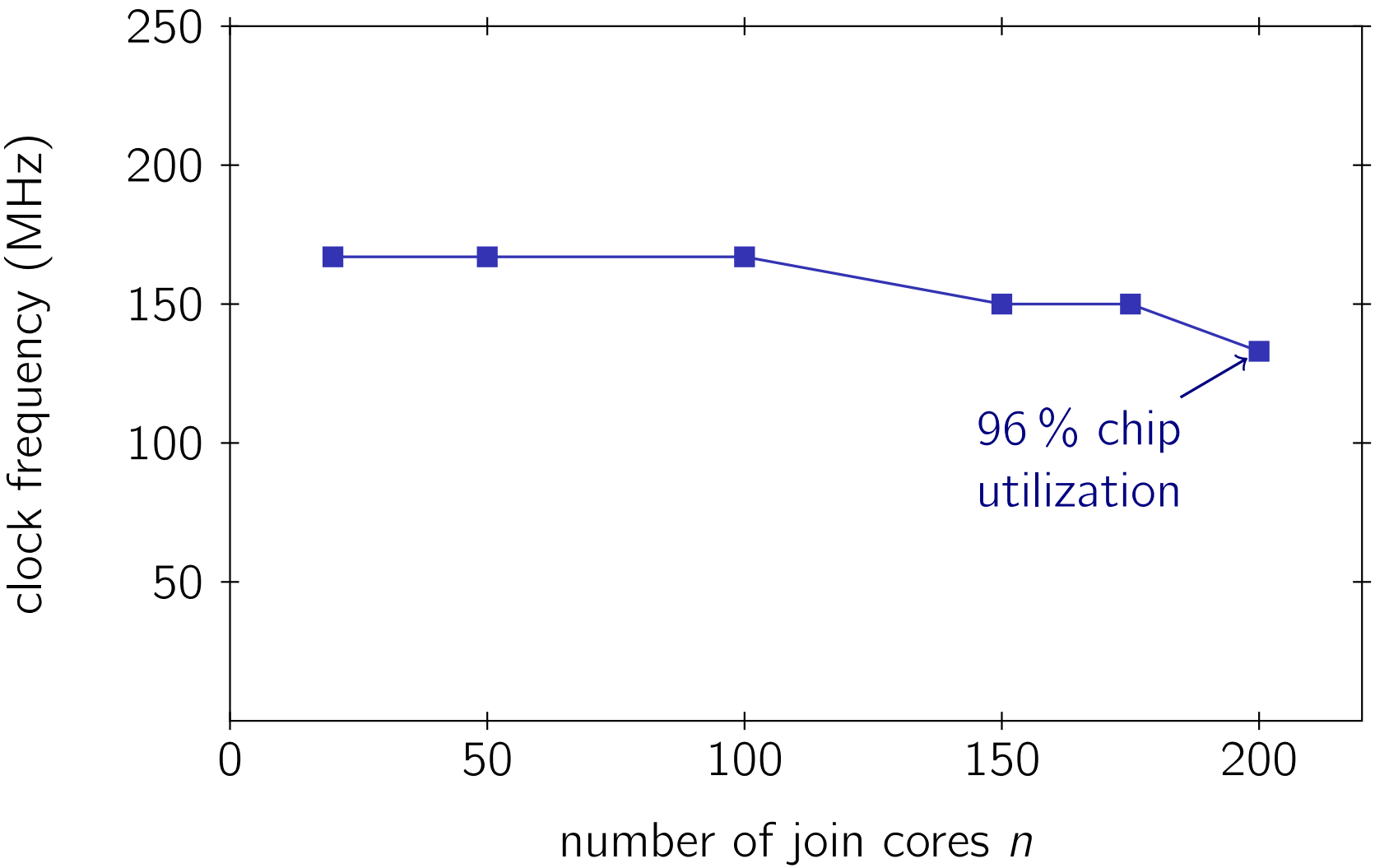
Example: AMD "Magny Cours" (48 cores)



Experiments (AMD Magny Cours, 2.2 GHz)



Beyond 48 Cores. . . (FPGA-based simulation)



Highly Concurrent Workloads

Databases are often faced with **highly concurrent workloads**.

Good news:

- Exploit parallelism offered by hardware (increasing number of cores).

Bad news:

- Increases relevance of **synchronization mechanisms**.

Two levels of synchronization in databases:

Synchronize on User Data
: to guarantee transaction semantics; database terminology: **locks**

Synchronize on Database-Internal Data Structures
: short-duration locks; called **latches** in databases

We'll now look at the latter, even when we say "locks."

Lock (Latch) Implementation

There are two strategies to implement locking:

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

Spinning (can be done in user space)

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

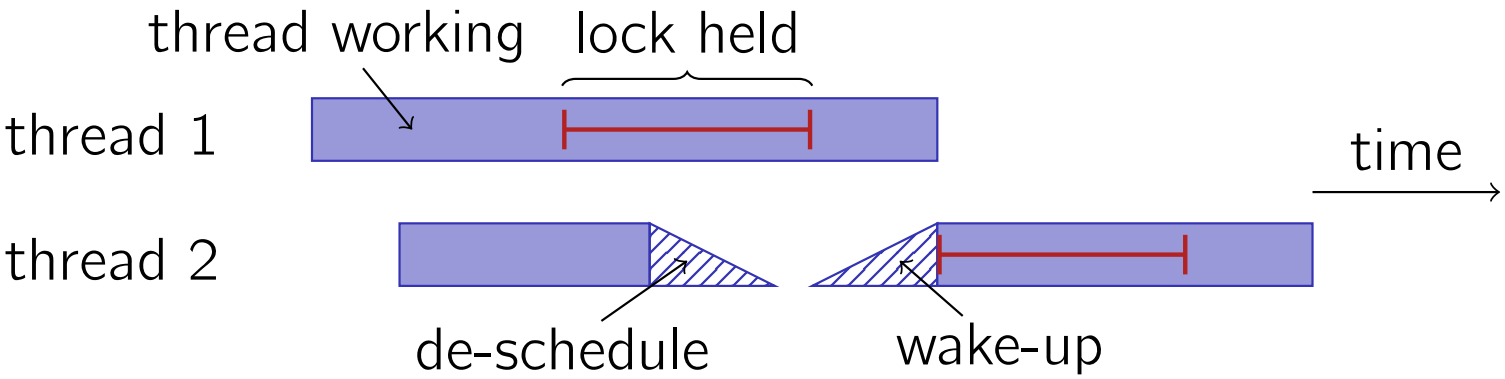
  $\rightarrow \approx 150$ nsec
- Thread burns CPU cycles while spinning.

Implementation of Spinlocks

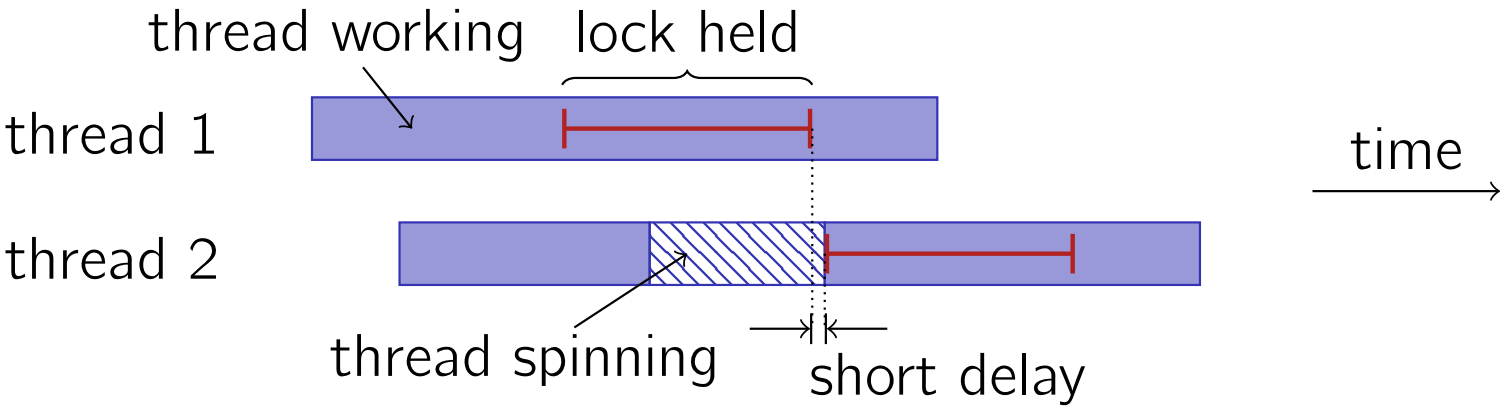
Implementation of a spinlock?

Thread Synchronization

Blocking:

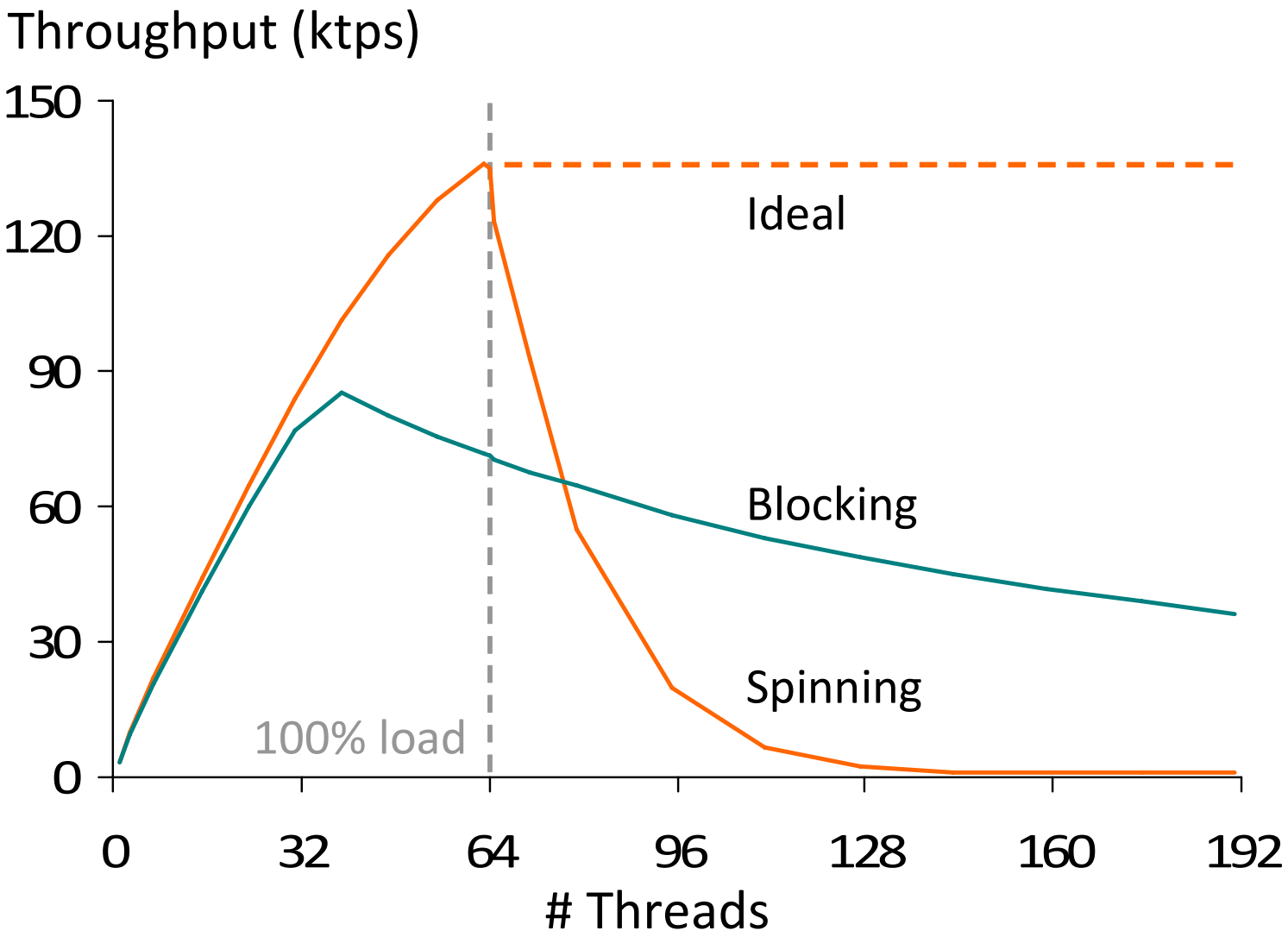


Spinning:



Experiments: Locking Performance

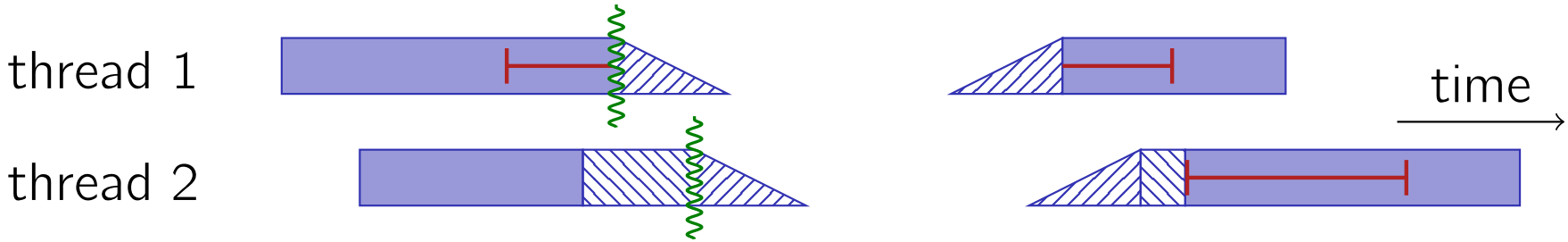
Sun Niagara II (64 hardware contexts):



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

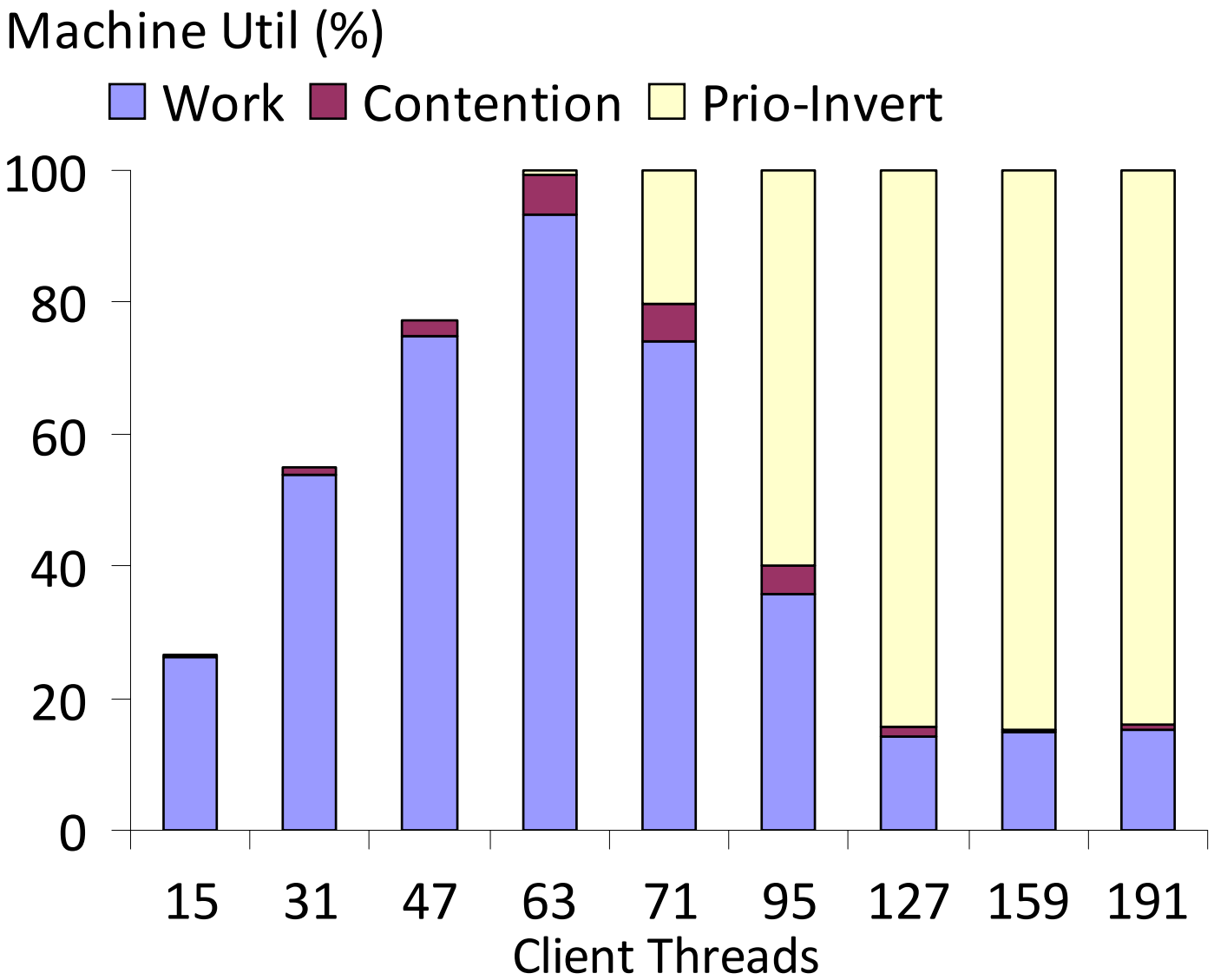
Spinning Under High Load

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.*

The Right Tool for the Right Purpose

The properties of spinning and blocking suggest their use for different purposes:

- **Spinning** features **quick lock hand-offs**.
  - → Use spinning to coordinate access to a shared data structure (contention).
- **Blocking** reduces **system load** ($\leadsto$ scheduling).
  - → Use blocking at longer time scales.
  - → Block when system load increases to reduce scheduling overhead.

Idea: Monitor system load (using a separate thread) and control spinning/blocking behavior off the critical code path.

Load Controller

The **load controller** periodically

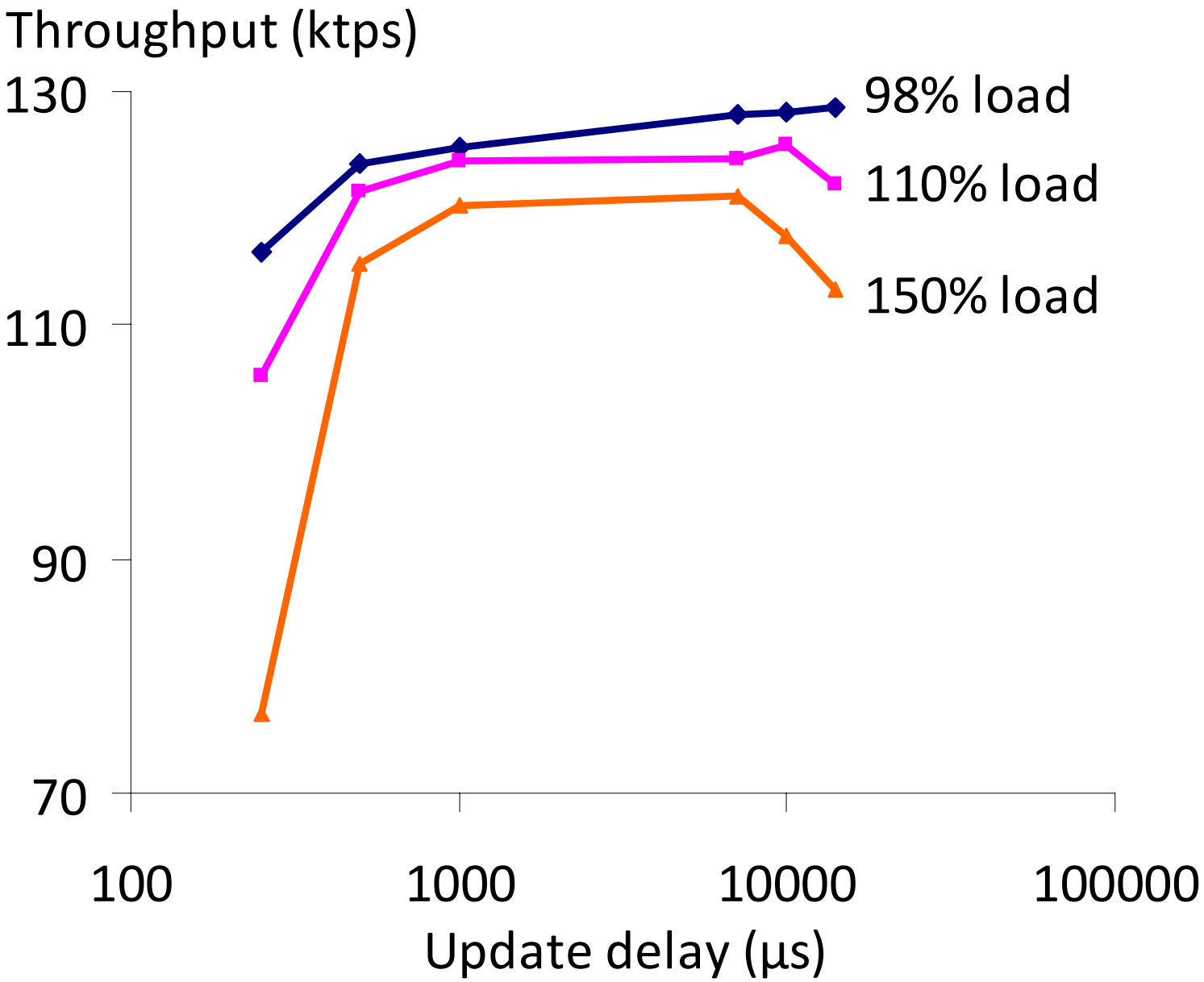
- Determines current load situation from the OS.
- If system gets **overloaded**
  - "invite" threads to block with help of a **sleep slot buffer**.
  - Size of sleep slot buffer: number of threads that should block.
- When load gets less
  - controller **wakes up** sleeping threads, which register in sleep slot buffer before going to sleep.

Lock Handling

A thread that wants to acquire a lock

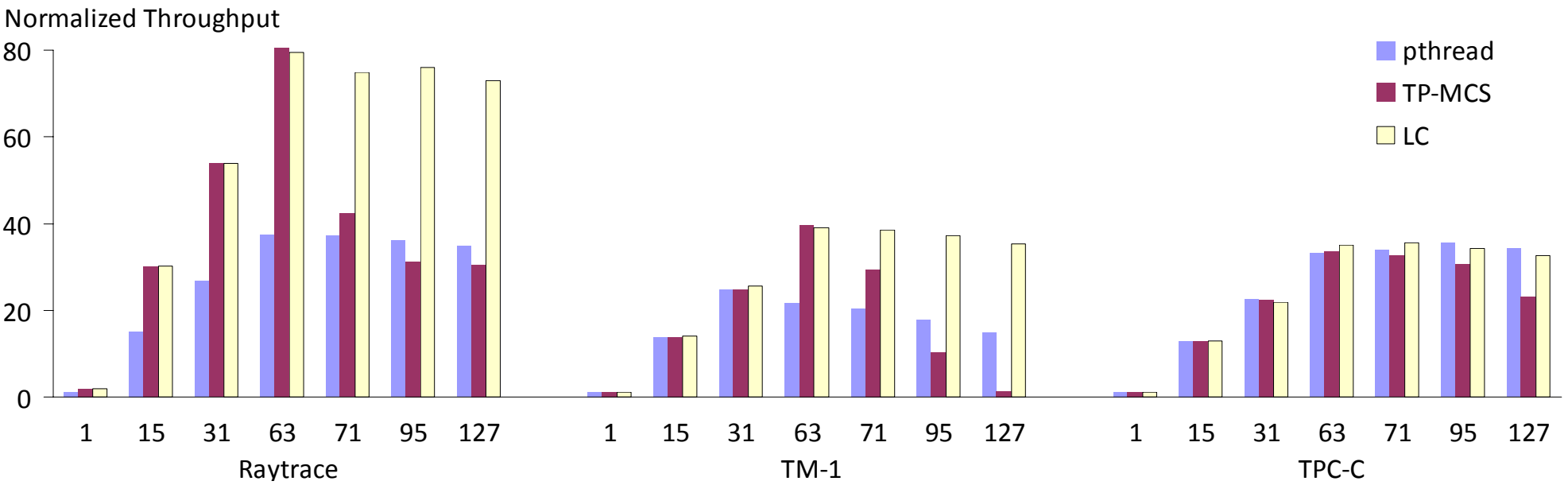
- Checks the regular **spin lock**.
- If the lock is already taken, it tries to enter the sleep slot buffer and blocks (otherwise it spins).
- The load controller will wake up the thread in time.

Controller Overhead



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

Performance Under Load



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