Data Processing on Modern Hardware

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

Summer 2014

Part V

Execution on Multiple Cores

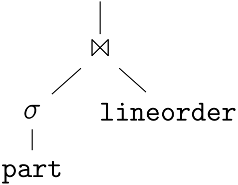
Example: Star Joins

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
```

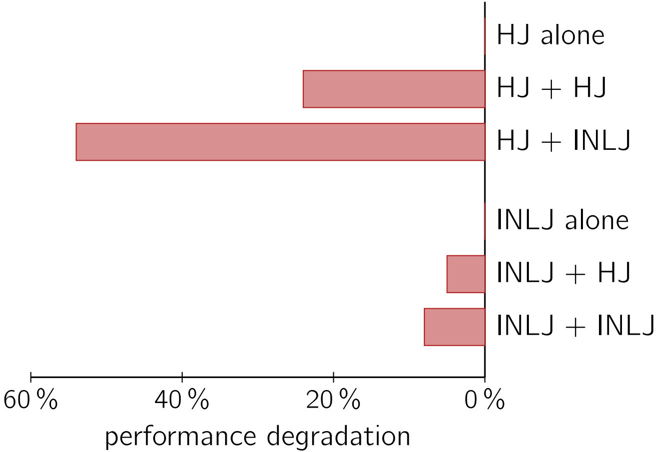


To implement ⋈ use either

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

Execution on "Independent" CPU Cores

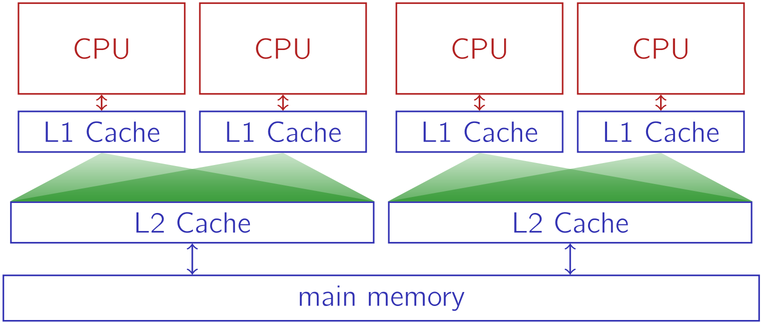
Co-run independent instances on different CPU cores.



Concurrent queries may seriously affect each other's performance.

Shared Caches

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

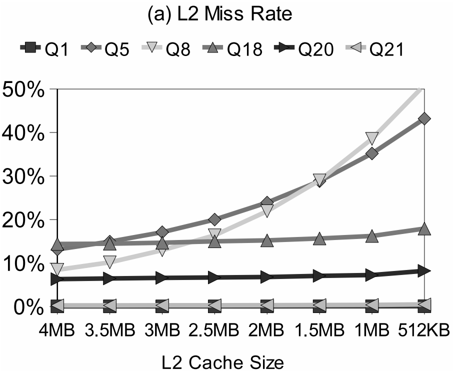


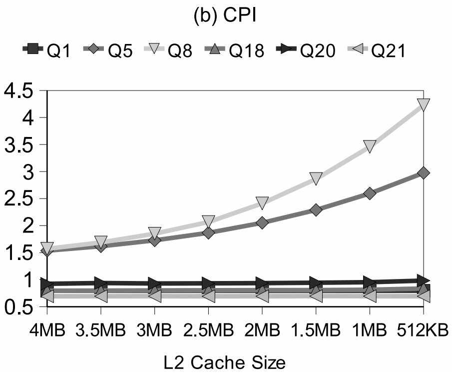
What we saw was **cache pollution**.

→ 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

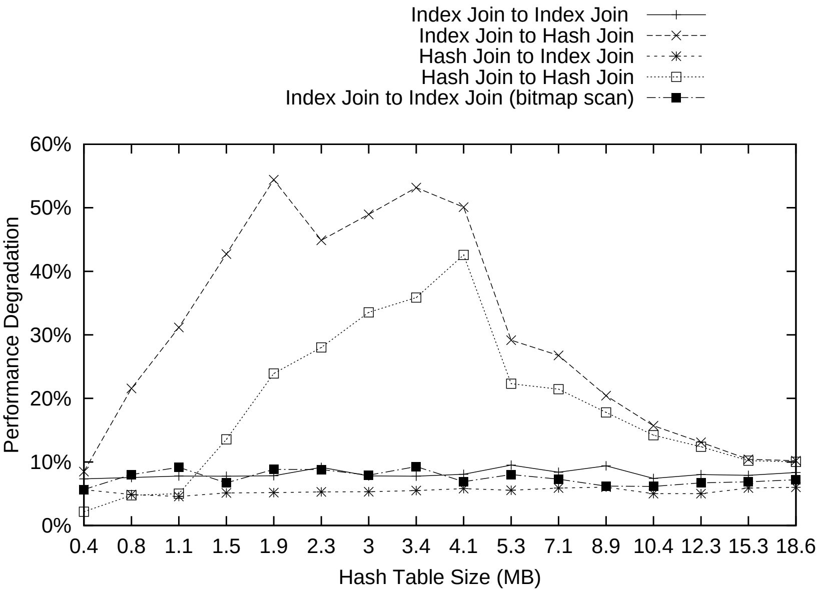
	strong	moderate	weak
amount of cache used	small	large	large
amount of cache needed	small	large	small

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:

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

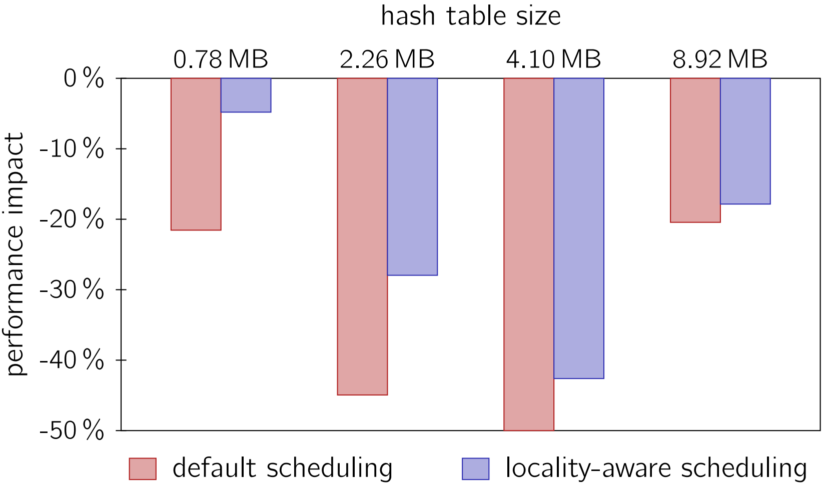
- **Co-schedule** queries to minimize (the impact of) cache pollution.

✎ **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:

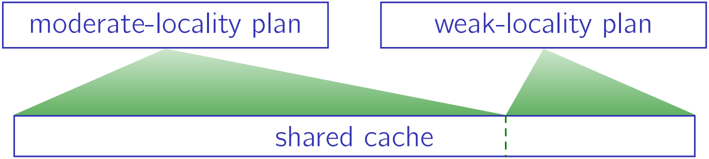


Source: Lee *et al.* *VLDB 2009*.

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.



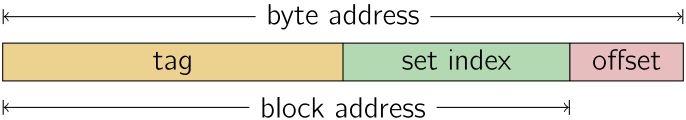
But:

- 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.



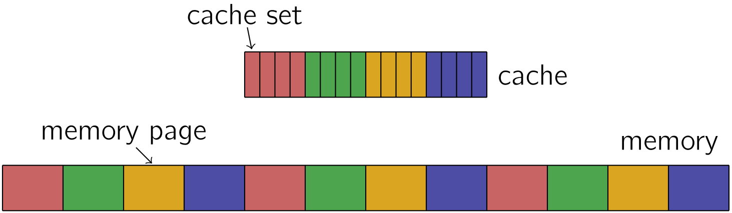
Thus,

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

Page Coloring

The address ↔ 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

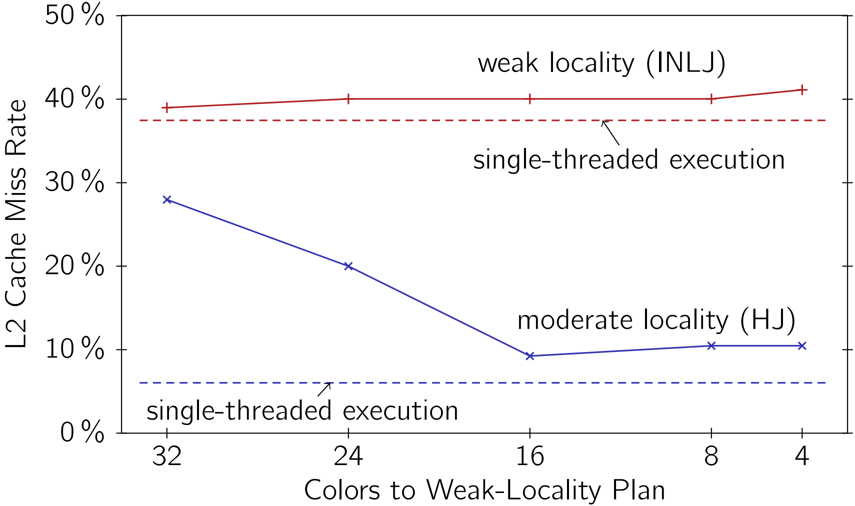
MCC-DB ("Minimizing Cache Conflicts"):

- 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

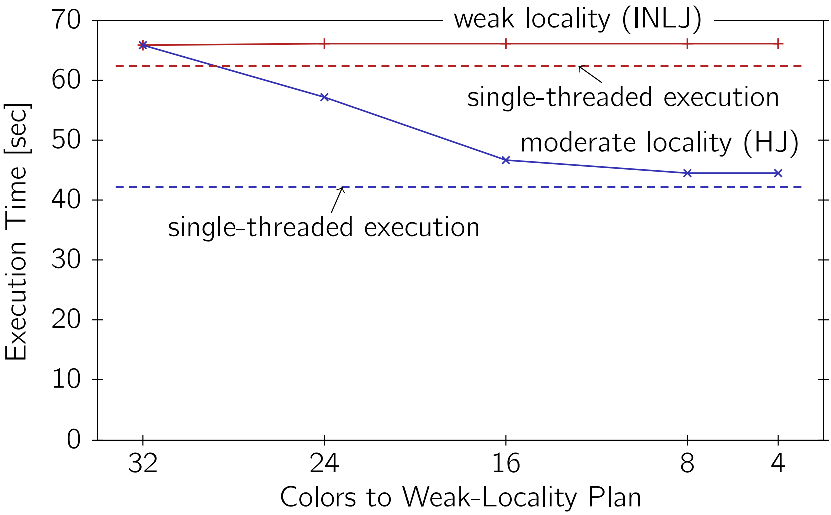
Moderate-locality hash join and weak-locality co-runner (INLJ):



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

Experiments

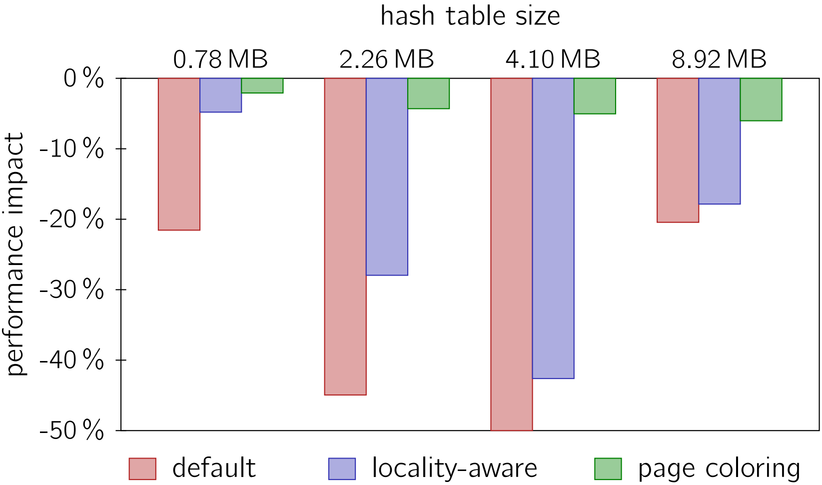
Moderate-locality hash join and weak-locality co-runner (INLJ):



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

Experiments: MCC-DB

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



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

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 μsec

Spinning (can be done in user space)

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

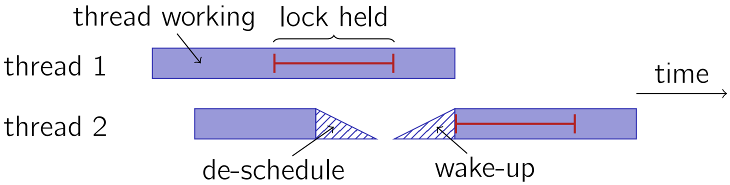
  $\rightarrow$ $\approx$ 150 nsec
- Thread burns CPU cycles while spinning.

Implementation of Spinlocks

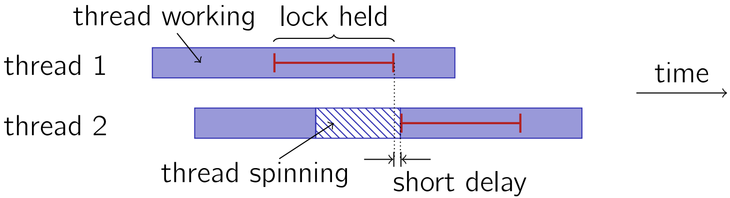
✎ **Implementation of a spinlock?**

Thread Synchronization

Blocking:

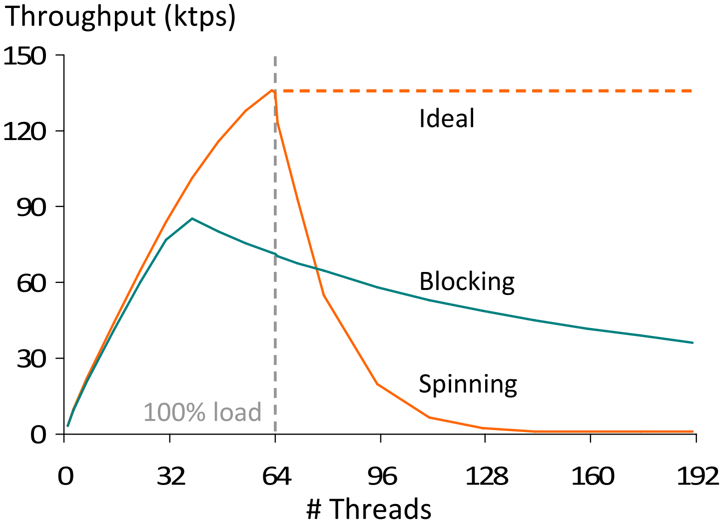


Spinning:



Experiments: Locking Performance

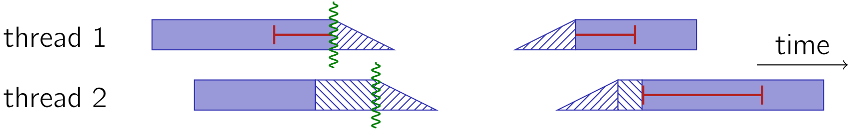
Sun Niagara II (64 hardware contexts):



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

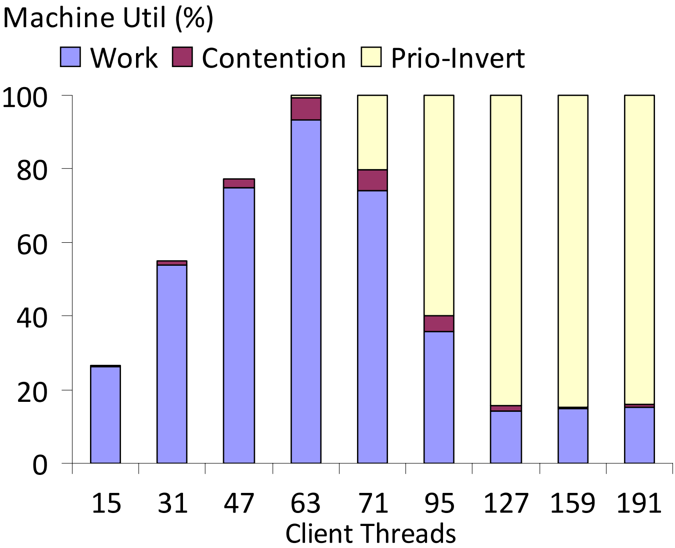
Spinning Under High Load

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 $\rightarrow$ 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** ($\rightsquigarrow$ 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

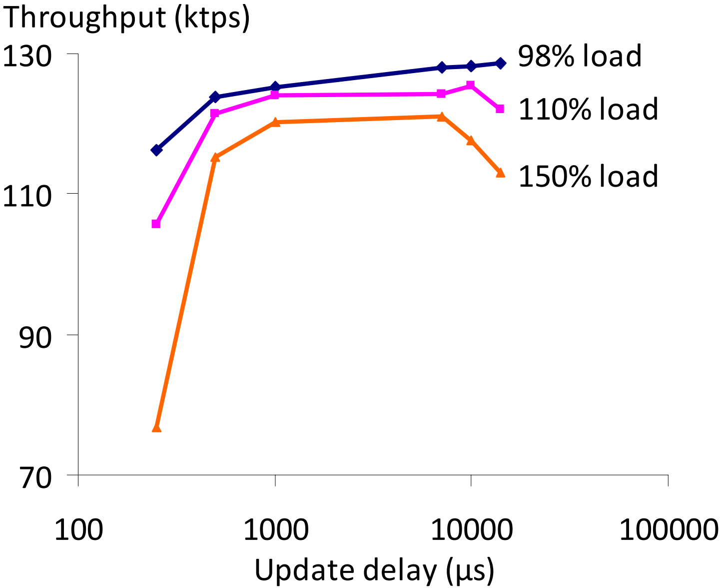
- Determines current load situation from the OS.
- If system gets **overloaded**
  - "invite" threads to block with help of a **sleep slot buffer**.
  - Size of sleep slot buffer: number of threads that should block.
- When load gets less
  - controller **wakes up** sleeping threads, which register in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

Lock Handling

A thread that wants to acquire a lock

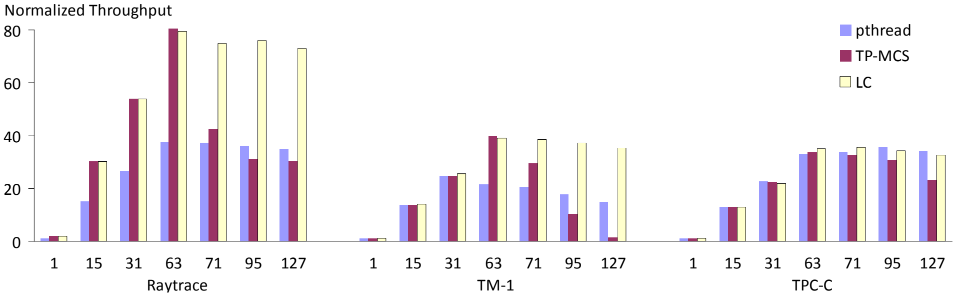
- Checks the regular **spin lock**.
- If the lock is already taken, it tries to enter the sleep slot buffer and blocks (otherwise it spins).
- The load controller will wake up the thread in time.

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*.