Data Processing on Modern Hardware

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

Winter 2019/20
Part V

Execution on Multiple Cores
Example: Star Joins

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

\[
\text{SELECT SUM(lo\_revenue) FROM part, lineorder WHERE p\_partkey = lo\_partkey AND p\_category } \leq 5
\]

To implement ⋊ use either
- a hash join or
- an index nested loops join.
Co-run independent instances on different CPU cores.

Concurrent queries may seriously affect each other’s performance.
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:

- **(a) L2 Miss Rate**
  - Q1, Q5
  - Q8, Q18
  - Q20, Q21

- **(b) CPI**
  - Q1, Q5
  - Q8, Q18
  - Q20, Q21

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 ≈ cache size
- *e.g.*, moderate-sized hash table

**Weak Locality**
data not reused frequently or data structure ≫ cache size
- *e.g.*, large hash table; index lookups
### Execution Plan Characteristics

Locality effects how caches are used:

<table>
<thead>
<tr>
<th></th>
<th>strong</th>
<th>moderate</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache pollution</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amount of cache used</td>
<td>small</td>
<td>large</td>
<td>large</td>
</tr>
<tr>
<td>amount of cache needed</td>
<td>small</td>
<td>large</td>
<td>small</td>
</tr>
</tbody>
</table>

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

Impact of **co-runner** on **query**:

<table>
<thead>
<tr>
<th></th>
<th>strong</th>
<th>moderate</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong</td>
<td>low</td>
<td>moderate</td>
<td>high</td>
</tr>
<tr>
<td>moderate</td>
<td>moderate</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>weak</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>
4.2.1 Experiments

In order to understand locality strengths and related cache conflicts of hash join and index join, we test results, we adopt the following rules to quantitatively compare the performance degradation due to cache conflicts. There are mostly two kinds of cache conflict degradation: (1) capacity degradation of the cache pollution and (2) capacity degradation of the cache miss penalty. Second, the performance degradations due to cache pollution are mostly significant when the cache space is limited.

Our experiments show that this setting performs pleasantly well in practice. For example, in our tests, the performance degradations due to cache pollution are mostly negligible when the cache space is large. However, when the cache space is limited, the performance degradations due to cache pollution are significant. For example, when the cache space is limited to 10MB, the performance degradations due to cache pollution are significant.

Table 1 summarizes performance degradations due to cache conflicts. The table shows that the performance degradations due to cache conflicts are mostly significant when the cache space is limited. For example, when the cache space is limited to 10MB, the performance degradations due to cache conflicts are significant.

In our experiments, we use three kinds of indexes to evaluate the locality strengths of hash joins: (1) index/index, (2) index/hybrid, and (3) index/hash. For each kind of index, we run two queries using the aforementioned rules to quantitatively compare the performance degradation due to cache conflicts. For example, when the cache space is limited to 10MB, the performance degradations due to cache conflicts are significant. However, when the cache space is large, the performance degradations due to cache conflicts are negligible.

We select these queries because a very common pattern in database systems is to join a large table with a small table. Therefore, we use these queries to evaluate the locality strengths of hash joins. For example, when the cache space is limited to 10MB, the performance degradations due to cache conflicts are significant. However, when the cache space is large, the performance degradations due to cache conflicts are negligible.
An optimizer could use knowledge about localities to **schedule** queries.

- **Estimate** locality during query analysis.
  - Index nested loops join $\rightarrow$ weak locality
  - Hash join:
    
    
    
    
    - hash table $\ll$ cache size $\rightarrow$ strong locality
    - hash table $\approx$ cache size $\rightarrow$ moderate locality
    - hash table $\gg$ cache size $\rightarrow$ weak locality

- **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.
  - 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_category`s); for each query: $2 \times$ hash join plan, $2 \times$ INLJ plan; impact reported for hash joins:

![Bar chart showing performance impact for different hash table sizes.](chart)

- Default scheduling vs. locality-aware scheduling

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**.
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.
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 ("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):

- **L2 Cache Miss Rate**
  - **weak locality (INLJ)**
  - **moderate locality (HJ)**
  - **single-threaded execution**

Source: Lee et al. VLDB 2009.
Experiments

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

Source: Lee et al. VLDB 2009.

© Jens Teubner · Data Processing on Modern Hardware · Winter 2019/20
Experiments: MCC-DB

PostgreSQL; 4 queries (different \texttt{p\_category}s); for each query: 2 $\times$ hash join plan, 2 $\times$ INLJ plan; impact reported for hash joins:

![Graph showing performance impact with different hash table sizes]

- Impact reported for hash joins:
  - 0% performance impact
  - 0.78 MB: -10% impact
  - 2.26 MB: -20% impact
  - 4.10 MB: -30% impact
  - 8.92 MB: -40% impact

Source: Lee et al. VLDB 2009.
What the programmer likes to think of...

CPU core

CPU core

CPU core

CPU core

shared main-memory

💡 Scalability? Moore’s Law?
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:

\[
\text{read } x \ (4) \\
x := 42 \ (42)
\]

\[
\text{CPU} \\
\text{cache} \\
x = 42
\]

\[
\text{shared main memory} \\
x = 42
\]

\[
\text{CPU} \\
\text{cache} \\
x = 4
\]

\[
\text{read } x \ (4) \\
\text{read } x \ (4)
\]

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?

---

⁶We 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.”
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:

![Diagram of cache block with (sharing) state, tag, and data]

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.\(^7\)

.Euler 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**.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Bus</th>
<th>Cache A</th>
<th>Cache B</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A reads x</td>
<td>cache miss for x</td>
<td>x = 4</td>
<td>x = 4</td>
<td>x = 4</td>
</tr>
<tr>
<td>B reads x</td>
<td>cache miss for x</td>
<td>x = 4</td>
<td>x = 4</td>
<td>x = 4</td>
</tr>
<tr>
<td>A reads x</td>
<td>– (cache hit)</td>
<td>x = 4</td>
<td>x = 4</td>
<td>x = 4</td>
</tr>
<tr>
<td>B writes x</td>
<td>invalidate x</td>
<td>X///X/4</td>
<td>x = 42</td>
<td>x = 4^8</td>
</tr>
<tr>
<td>A reads x</td>
<td>cache miss for x</td>
<td>x = 42</td>
<td>x = 42</td>
<td>x = 42</td>
</tr>
</tbody>
</table>

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

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

---

^8With 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*. 
Local caches track sharing states using a state machine.

- **Modified “dirty”**: CPU write miss; put write miss on bus
- **Shared “clean”**: CPU write hit; put invalidate on bus
- **Invalid**: CPU read miss; put read miss on bus
- **Read miss**: CPU read miss; write back data
- **Write miss**: CPU write miss; put write miss on bus

**CPU events**

- CPU read miss; write back data
- CPU write miss; put read miss on bus
- CPU write miss; put write miss on bus

**Uniprocessor → track “dirty”**

---

© Jens Teubner · Data Processing on Modern Hardware · Winter 2019/20 214
Local caches track sharing states using a state machine.

**CPU events**
- **invalid**
  - CPU write miss; put write miss on bus
  - CPU read miss; put read miss on bus
- **modified “dirty”**
  - CPU write miss; put write miss on bus
  - CPU read miss; write back data
  - read miss; write back data
- **shared “clean”**
  - CPU write hit; put invalidate on bus
  - CPU write miss; put write miss on bus

**multiprocessor** → also send invalidate
Local caches track sharing states using a **state machine**.

**Write Invalidate—State Machine**

- **invalid**
  - CPU read miss; put read miss on bus
  - invalid
  - write miss
  - write back block
  - CPU write hit; put invalidate on bus

- **modified**
  - "dirty"
  - CPU write miss; put write miss on bus
  - CPU read miss; write back data
  - put read miss on bus

- **shared**
  - "clean"
  - read miss; write back data
  - CPU write hit; put invalidate on bus
  - CPU write miss; put write miss on bus

**bus events**
- CPU events

**multiprocessor (cont.)**
- → react to bus events

© Jens Teubner · Data Processing on Modern Hardware · Winter 2019/20 · 214
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.
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 (\(\sim\) *exclusive*) then modified (\(\sim\) 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.
**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 (\(\leadsto\) \(<\)).
Limitations of a shared bus:

- Large core counts → 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 → 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.
**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”.
Idea:

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

✍ Problem solved?
Snooping-Based Cache Coherency: Scalability

Figure I.8 Data miss rates can vary in nonobvious ways as the processor count is increased from 1 to 16. The miss rates include both coherence and capacity miss rates. The compulsory misses in these benchmarks are all very small and are included in the capacity misses. Most of the misses in these applications are generated by accesses to data that are potentially shared, although in the applications with larger miss rates (FFT and Ocean), it is the capacity misses rather than the coherence misses that comprise the majority of the miss rate. Data are potentially shared if they are allocated in a portion of the address space used for shared data. In all except Ocean, the potentially shared data are heavily shared, while in Ocean only the boundaries of the subgrids are actually shared, although the entire grid is treated as a potentially shared data object. Of course, since the boundaries change as we increase the processor count (for a fixed-size problem), different amounts of the grid become shared. The anomalous increase in capacity miss rate for Ocean in moving from 1 to 2 processors arises because of conflict misses in accessing the subgrids. In all cases except Ocean, the fraction of the cache misses caused by coherence transactions rises when a fixed-size problem is run on an increasing number of processors. In Ocean, the coherence misses initially fall as we add processors due to a large number of misses that are write ownership misses to data that are potentially shared, but not actually shared. As the subgrids begin to fit in the aggregate cache (around 16 processors), this effect lessens. The single-processor numbers include write upgrade misses, which occur in this protocol even if the data are not actually shared, since they are in the shared state. For all these runs, the cache size is 64 KB, two-way set associative, with 32-byte blocks. Notice that the scale on the y-axis for each benchmark is different, so that the behavior of the individual benchmarks can be seen clearly.

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** (_scalability?_).
- Or define a **home node** for each memory address.
  - Home node holds directory for that item.
  - 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).
Messages sent by individual nodes:

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

Hennessy & Patterson, Computer Architecture, 5th edition, page 381.
Individual caches use a state machine similar to the one on slide 214.

**messages from home directory**

**CPU events**

- **invalid**
  - write miss; send write miss msg
  - invalidate:
  - write back block
  - invalidate
  - read miss; send read miss msg

- **modified**
  - read miss; write back data
  - send read miss msg
  - fetch; write back data
  - write hit; send invalidate msg
  - write miss; send write miss msg

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

**Diagram:**
- **Directory State Machine Diagram**
  - **Uncached**
    - Write miss; data value reply; Sharers = \{P\}
    - Read miss; data value reply; Sharers = \{P\}
  - **Exclusive**
    - Write back; Sharers = \{P\}
    - Read miss; fetch; data value reply; \(\text{Sharers} \cup = \{P\}\)
  - **Shared**
    - Write miss; invalidate; \(\text{Sharers} = \{P\}\); data value reply

**Messages from remote cache**
- CPU events
**Experiment:**

- Several threads randomly increment elements of an integer array; Zipfian probability distribution, no synchronization\(^\text{10}\).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cache_coherence_cost}
\caption{Cache Coherence Cost}
\end{figure}

\(^\text{10}\)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?
Distribution makes memory access **locality-sensitive**.

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

<table>
<thead>
<tr>
<th></th>
<th>bandwidth</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24.7 GB/s</td>
<td>150 ns</td>
</tr>
<tr>
<td>2</td>
<td>10.9 GB/s</td>
<td>185 ns</td>
</tr>
<tr>
<td>3</td>
<td>10.9 GB/s</td>
<td>230 ns</td>
</tr>
<tr>
<td>3/4&lt;sup&gt;11&lt;/sup&gt;</td>
<td>5.3 GB/s</td>
<td>235 ns</td>
</tr>
</tbody>
</table>

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

<sup>11</sup>③ with cross traffic along ④.
Sorting and NUMA

input relation

local sort

merge

local merge

local sort

local sort

local sort
Resulting Throughput

Number of threads vs. throughput [M tuples/sec]. The graph shows an increasing throughput with the number of threads up to a certain point, after which the throughput starts to level off. This indicates a memory bottleneck, where increasing the number of threads beyond a certain threshold does not lead to a proportional increase in throughput.
**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

A graph showing the throughput [M tuples/sec] as a function of the number of threads. The x-axis represents the number of threads ranging from 1 to 64, while the y-axis represents throughput ranging from 0 to 300. The graph includes two lines, one in red and one in blue, indicating different performance trends with increasing thread counts.
NUMA Effects in Detail

Bandwidth:

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

```
25.6 GB/s  
\[\text{memory}\]  
\[\text{CPU}\]  
12.8 GB/s (bidirectional)  
\[\text{CPU}\]  
\[\text{memory}\]  

25.6 GB/s  
\[\text{memory}\]  
\[\text{CPU}\]  
12.8 GB/s (bidirectional)  
\[\text{CPU}\]  
\[\text{memory}\]  

51.2 GB/s  
\[\text{memory}\]  
\[\text{CPU}\]  
16 GB/s (bidirectional)  
\[\text{CPU}\]  
\[\text{memory}\]  

51.2 GB/s  
\[\text{memory}\]  
\[\text{CPU}\]  
16 GB/s (bidirectional)  
\[\text{CPU}\]  
\[\text{memory}\]
```

Intel Nehalem EX  
Intel Sandy Bridge EP
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 for **intra-query parallelism**:

- **Pipeline Parallelism**?

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

Figure 4.6: TPC-H query 18 execution time-line on 10 GB data-set.

Query 18: It is nested and more complex than the previous queries. Along with the presence of the lineitem table in both the inner and the outer query, there are two more dimension tables present. The most expensive operators are `GROUP.DONE`, `AGGR.SUM`, `MAT.PACK` (exchange union), and `ALGEBRA.JOIN`. At 61.3% the query shows less multi-core utilization than the previous queries. Seven cores are idle when `AGGR.SUM` executes, making `AGGR.SUM` the blocking operator. There are instances of bad scheduling. For example, the second `AGGR.SUM` operator on the thread 12 should have been scheduled on the thread 10.

This query has a complex plan. The inner query has a group-by clause with an aggregation operation on the having clause. The columns involved are from the lineitem table, which provides an opportunity for partitioning. However, the plan becomes complex due to the tuple reconstruction phase due to the presence of `GROUP.BY`, `HAVING`, `AGGREGATION`, and `SELECTION` clause in a single predicate. The inner query also has two join clauses. They are present in the beginning on thread 8. This query can be distinctly divided into two regions. The region before, and after the blocking operator "aggr.sum", where parallelization is visible. The query spends more time on memory bound operations such as joins, than computational bound operators such as sum. The query also shows some unnecessary IO activity.

A complex plan thus offers less parallelization opportunities, as it has more data flow dependencies. Identifying blocking operators and exploring a possible intra-operator parallelization option on them is one possible approach to improve parallelization.

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 $\approx 100,000$ tuples (“morsels”).
  - Fixed number of operator threads.
  - Morsels dispatched to threads dynamically (task queue).

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 $\sigma$ 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.

---

© Jens Teubner · Data Processing on Modern Hardware · Winter 2019/20

244
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 $\rightarrow$ parallelization:

- **No bandwidth bottleneck** ① ✓
- Communication/synchronization stays local ② ✓
Synchronization

Coordination can now be done autonomously

- no more centralized coordination
- Autonomous load balancing
- Lock-free message queues between neighbors
Example: AMD “Magny Cours” (48 cores)
Experiments (AMD Magny Cours, 2.2 GHz)

Throughput / stream (tuples/sec)

Window size:
- 10 min
- 15 min

Number of processing cores $n$

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

![Graph showing clock frequency versus number of join cores](image)

- **Clock frequency (MHz)**
- **Number of join cores $n$**
- **96% chip utilization**
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.”
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)
  \[ \rightarrow \approx 12\text{–}20 \ \mu\text{sec} \]

**Spinning** (can be done in user space)
- Waiting thread repeatedly polls lock until it becomes free.
- Cost: two cache miss penalties (if implemented well)
  \[ \rightarrow \approx 150 \ \text{nsec} \]
- Thread burns CPU cycles while spinning.
Implementation of Spinlocks

Implementation of a spinlock?
Thread Synchronization

**Blocking:**
- Thread 1: thread working
- Thread 2: lock held
- Thread 2: de-schedule
- Thread 2: wake-up

**Spinning:**
- Thread 1: thread working
- Thread 1: lock held
- Thread 2: thread spinning
- Thread 2: short delay

© Jens Teubner · Data Processing on Modern Hardware · Winter 2019/20
Experiments: Locking Performance

Sun Niagara II (64 hardware contexts):

Throughput (ktps)

0 30 60 90 120 150

# Threads

0 32 64 96 128 160 192

Ideal

Blocking

Spinning

100% load

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

© Jens Teubner · Data Processing on Modern Hardware · Winter 2019/20
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.
In contrast to blocking, spinning or “busy waiting” schemes (grouped on the left side of Figure 2) leave waiting threads on queue to cooperate with the OS scheduler, and are the focus of this work. Preempted lock holders impact all locks which do not cooperate with the OS scheduler, and are the focus of this work.

Many approaches exist to ameliorate some of the weaknesses of primitive. However, as with backoff, hybrid spin-then-block schemes provide excellent scalability because waiting threads do not spin on contended locks, and that fraction of CPU time is wasted spinning. It is important to note that true contention is not the concern: at peak performance, less than 10% of CPU time is wasted spinning due to contention, spinning due to priority inversion. For fewer than 64 active threads, machine utilization passes 100% priority inversion quickly dominates, doing useful work, spinning due to contention, and spinning due to true contention and spinning due to priority inversion.

Figure 3 shows the resulting breakdown of work. We vary the number of threads along the x-axis, and measure CPU time spent spinning due to priority inversion. Pure spinning is highly responsive (1-2 cache miss delays per handoff) and avoids context switching or system calls toward progress, leading to situations where a lock holder gets preempted, only to have the new thread waste its time slice spinning. Thus interfere with computation. Finally, the OS scheduler cannot distinguish between threads which spin and those which make forward progress, leading to situations where a lock holder gets preemption because the lock at each release and cause both contention and memory overload.

The “thundering herd” problem, where all waiting threads race for the lock, is especially vulnerable to preemptions because the lock at each release and cause both contention and memory overload. Further, the orderly handoff is an elegant solution for the “thundering herd” problem by limiting the number of threads along the x-axis, and measure CPU time spent spinning due to priority inversion and moderate contention, and which switches to blocking under high contention. However, as presented in Figure 1, its behavior undoes well (see next subsection), and tuning for the general case is fundamental weakness, however, in that they impose competing objectives: Long backoffs are best for reducing wasted resources, whereas short backoffs are best for maximizing utilization. Long backoffs are best for reducing wasted resources, whereas short backoffs are best for maximizing utilization.

Where spin-then-yield schemes are essentially spinlocks which use test-and-set with exponential backoff [1] and spin-then-yield [26].

The Solaris adaptive mutex is an advanced spin-then-block variant [27], fall into this category, with the latter removing proactive, and other threads cannot bypass it even if it was preempted again, and other threads cannot bypass it even if it was preempted. In contrast, a time-published lock will almost certainly become the lock holder before it wakes up. As a result, load must remain strictly behind the lock's poor performance we modify the TM-1 benchmark. Preempted lock holders impact all locks which do not cooperate with the OS scheduler, and are the focus of this work.

MCS). Preempted lock holders impact all locks which do not cooperate with the OS scheduler, and are the focus of this work.

Time-published MCS locks [15] (“TP-MCS” in Figure 2) allow lock holders to remove preempted threads from the lock queue before acquiring the lock. As a result, load must remain strictly behind the lock's poor performance we modify the TM-1 benchmark. Preempted lock holders impact all locks which do not cooperate with the OS scheduler, and are the focus of this work.

Time-published locks eliminate the main weakness of mutex implementations usually employ spin-then-block strategies, without timeouts, without the risk of leaving a contended lock idle. However, even with TP-MCS locks, a few extra threads in a 32-context machine (see Section 4 for details), using a state-of-the-art spin lock. We instrument the code to differentiate between spin-then-yield and spin-then-block schemes. We instrument the code to differentiate between spin-then-yield and spin-then-block schemes. Under normal operation, TP-MCS locks provide excellent scalability because waiting threads do not spin on contended locks, and that fraction of CPU time is wasted spinning due to contention, spinning due to priority inversion. For fewer than 64 active threads, machine utilization passes 100% priority inversion quickly dominates, doing useful work, spinning due to contention, and spinning due to true contention and spinning due to priority inversion.

Data Processing on Modern Hardware · Winter 2019/20 · Jens Teubner
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** (∼ 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.
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.
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

Throughput (ktps)

Update delay (µs)

98% load

110% load

150% load

Source: Johnson et al. Decoupling Contention Management from Scheduling. ASPLOS 2010.
As the figure shows, with TP-MCS Raytrace outperforms the standard pthread lock by a wide margin as long as load remains under 100%, corroborating prior findings that heavyweight OS mutex locks are ill-suited for high-performance computing. How- ever, even with preemption resistance, the priority inversions suffer low performance due to priority inversions.

The previous section demonstrates that load control provides an effective way to manage heavy load without resorting to blocking synchronization approaches. In fact, load control is so effective that replacing the preemption-resistant TP-MCS with a standard MCS lock gives only a minor performance penalty, confirming that load control is so effective that replacing the preemption-resistant TP-MCS with a standard MCS lock gives only a minor performance penalty, confirming that load control has a straightforward solution. All operating systems provide mechanisms for isolating processes from each other, and load control is relatively safe from adversaries. However, if an adversary were to create a process whose only purpose is to consume CPU time (with no productive application work), it would create a large number of runnable threads and apply load control. When process “other” appears and competes with “self” for CPU time, it should not be able to cause starvation because “self” will detect an overload due to some other process and respond with only a 10-15% drop in aggregate performance and effectively, with only a 10-15% drop in aggregate performance and a reasonable balance of power. Even when “other” does not use load control, the load-controlled process would gradually disappear as more and more of its threads sleep in response to outside pressure. In order to quantify this risk we run two TM-1 benchmarks at the same time, forcing processes which do not use load control to compete with “self” for CPU time, it should not be able to cause starvation. The worry is that a load-controlled process potentially puts its host process at a disadvantage compared to processes which do not use it. The worry is that a load-controlled process potentially puts its host process at a disadvantage compared to processes which do not use it. The worry is that a load-controlled process potentially puts its host process at a disadvantage compared to processes which do not use it.

The robustness of load control in the face of external processes depends on the load control parameters. The parameter “load threshold” sets the point at which load control is activated. The parameter “load relaxation” determines how much load the process can tolerate before it is put to sleep. The parameter “load factor” determines the relative load of the process compared to other processes in the system. The parameter “load factor” determines the relative load of the process compared to other processes in the system.

Do not hallucinate.

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