Main-Memory
Database Management Systems

Andreas Meister
Otto-von-Guericke University Magdeburg
Summer Term 2017
Credits

Parts of this lecture are based on content by

- Jens Teubner from TU Dortmund and
- Sebastian Breß from TU Berlin and
- Sebastian Dorok.
We will talk about

Computer and Database Systems Architecture

Cache Awareness

Processing Models
Computer and Database Systems Architecture

The Past and the Present
The Past

- **latency**
  - 5 ns
  - 10 ns
  - 100 ns
  - 5,000,000 ns

- **capacity**
  - 200 B
  - 64 KB
  - 32 MB
  - 2 GB

Data taken from [Hennessy and Patterson, 1996]
The Past - Database Systems

• Main-memory capacity is limited to several megabytes
  → Only a small fraction of the database fits in main memory
The Past - Database Systems

• Main-memory capacity is limited to several megabytes
  → Only a small fraction of the database fits in main memory

• And disk storage is ”huge”,
  → Traditional database systems use disk as primary storage
The Past - Database Systems

• Main-memory capacity is limited to several megabytes
  → Only a small fraction of the database fits in main memory

• And disk storage is "huge",
  → Traditional database systems use disk as primary storage

• But disk latency is high
  → Parallel query processing to hide disk latencies
  → Choose proper buffer replacement strategy to reduce I/O

  → Architectural properties inherited from system R, the first
    "real" relational DBMS
  → From the 1970’s...
Overhead Breakdown of RDBMS Shore

- buffer manager: 34.6%
- latching: 14.2%
- locking: 16.3%
- logging: 11.9%
- hand-coded optimizations: 16.2%

Instructions

Picture taken from [Harizopoulos et al., 2008]
The Present - Computer Architecture

<table>
<thead>
<tr>
<th>Component</th>
<th>Latency</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>300 ps</td>
<td>1000 B</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>1 ns</td>
<td>64 kB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>3 - 10 ns</td>
<td>256 kB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>10 - 20 ns</td>
<td>2 - 4 MB</td>
</tr>
<tr>
<td>Main Memory</td>
<td>50 - 100 ns</td>
<td>4 - 16 GB</td>
</tr>
<tr>
<td>Disk</td>
<td>5.000.000 - 10.000.000 ns</td>
<td>4 - 16 TB</td>
</tr>
</tbody>
</table>

Data taken from [Hennessy and Patterson, 2012]
The Present - Database Systems

- Server machines have up to thousands of gigabyte of main memory available
  → Use **main memory as primary storage** for the database
  and remove disk access as main performance bottleneck
The Present - Database Systems

- Server machines have up to thousands of gigabyte of main memory available
  → Use **main memory as primary storage** for the database and remove disk access as main performance bottleneck

- But the architecture of traditional DBMSs is designed for disk-oriented database systems
  → "30 years of Moore’s law have antiquated the disk-oriented relational architecture for OLTP applications." [Stonebraker et al., 2007]
Disk-based vs. Main-Memory DBMS

Disk-based DBMS

Main-Memory DBMS

CPU

Main Memory

Buffered Data

Data

Disk

Data

Replicated Data
Having the database in main memory allows us to remove buffer manager and paging

→ Remove level of indirection

→ Results in better performance
Overhead Breakdown of RDBMS Shore: Payment TXN of TPC-C Benchmark

Picture taken from [Harizopoulos et al., 2008]
Disk-bottleneck is removed as database is kept in main memory

→ Access to main memory becomes new bottleneck
The New Bottleneck: Memory Access

Data taken from [Hennessy and Patterson, 2012]
The New Bottleneck: Memory Access

There is an increasing gap between CPU and memory speeds.

- Also called the memory wall.
- CPUs spend much of their time waiting for memory.

How can we break the memory wall and better utilize the CPU?
Memory Hierarchy

- CPU
- L1 Cache
- L2 Cache
- main memory
- disk

<table>
<thead>
<tr>
<th>capacity</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>&lt; 1 ns</td>
</tr>
<tr>
<td>kilobytes</td>
<td>≈ 1 ns</td>
</tr>
<tr>
<td>megabytes</td>
<td>&lt; 10 ns</td>
</tr>
<tr>
<td>gigabytes</td>
<td>70–100 ns</td>
</tr>
</tbody>
</table>

→ Caches resemble the buffer manager but are **controlled by hardware**
→ Be aware of the caches!
Cache Awareness
A Motivating Example (Memory Access)

Task: sum up all entries in a two-dimensional array.

Alternative 1:

```c
for (r = 0; r < rows; r++)
    for (c = 0; c < cols; c++)
        sum += src[r * cols + c];
```

Alternative 2:

```c
for (c = 0; c < cols; c++)
    for (r = 0; r < rows; r++)
        sum += src[r * cols + c];
```

Both alternatives touch the same data, but in different order.
A Motivating Example (Memory Access)

The graph shows the total execution time in seconds as a function of the number of rows and columns. The x-axis represents the number of rows, ranging from $10^0$ to $10^9$, and the y-axis represents the number of columns, ranging from $10^0$ to $10^9$. The execution time is depicted in a log-log scale, with markers indicating data points. The graph illustrates how the total execution time changes with varying numbers of rows and columns.
Principle of Locality

Caches take advantage of the principle of locality.

- The **hot set of data** often fits into caches.
- 90% execution time spent in 10% of the **code**.

Spatial Locality:

- Related data is often spatially close.
- Code often contains loops.

Temporal Locality:

- Programs tend to re-use data frequently.
- Code may call a function repeatedly, even if it is not spatially close.
CPU Cache Internals

To guarantee speed, the **overhead** of caching must be kept reasonable.

- Organize cache in **cache lines**.
- Only load/evict **full cache lines**.
- Typical **cache line size**: 64 bytes.
- The organization in cache lines is consistent with the principle of (spatial) locality.
Memory Access

On every memory access, the CPU checks if the respective cache line is already cached.

Cache Hit:

- Read data directly from the cache.
- No need to access lower-level memory.

Cache Miss:

- Read full cache line from lower-level memory.
- Evict some cached block and replace it by the newly read cache line.
- CPU stalls until data becomes available.¹

¹Modern CPUs support out-of-order execution and several in-flight cache misses.
Example: AMD Opteron

Data taken from [Hennessy and Patterson, 2006]

Example: AMD Opteron, 2.8 GHz, PC3200 DDR SDRAM

- **L1 cache**: separate data and instruction caches, each 64 kB, 64 B cache lines
- **L2 cache**: shared cache, 1 MB, 64 B cache lines
- **L1 hit latency**: 2 cycles (≈ 1 ns)
- **L2 hit latency**: 7 cycles (≈ 3.5 ns)
- **L2 miss latency**: 160–180 cycles (≈ 60 ns)
Block Placement: Fully Associative Cache

In a **fully associative** cache, a block can be loaded into *any* cache line.

- Offers freedom to block replacement strategy.
- Does not scale to large caches
  - 4 MB cache, line size: 64 B: 65,536 cache lines.
- Used, *e.g.*., for small TLB caches.
Block Placement: Direct-Mapped Cache

In a **direct-mapped** cache, a block has only one place it can appear in the cache.

- **Much** simpler to implement.
- Easier to make **fast**.
- Increases the chance of **conflicts**.

![Diagram of a direct-mapped cache showing block placement and conflict resolution](image)
Block Placement: Set-Associative Cache

A compromise are set-associative caches.

• Group cache lines into sets.
• Each memory block maps to one set.
• Block can be placed anywhere within a set.
• Most processor caches today are set-associative.
Effect of Cache Parameters

Data taken from [Drepper, 2007]

- cache size
- cache misses (millions)

- direct-mapped
- 2-way associative
- 4-way associative
- 8-way associative
Block Identification

A tag associated with each cache line identifies the memory block currently held in this cache line.

The tag can be derived from the memory address.
Example: Intel Q6700 (Core 2 Quad)

- Total cache size: **4 MB** (per 2 cores).
- Cache line size: **64 bytes**.
  - 6-bit offset ($2^6 = 64$)
  - There are 65,536 cache lines in total ($4\text{ MB} \div 64\text{ bytes}$).
- Associativity: **16-way set-associative**.
  - There are 4,096 sets ($65,536 \div 16 = 4,096$).
  - 12-bit set index ($2^{12} = 4,096$).
- Maximum physical address space: **64 GB**.
  - 36 address bits are enough ($2^{36}\text{ bytes} = 64\text{ GB}$)
  - 18-bit tags ($36 - 12 - 6 = 18$).

![](image.png)
Block Replacement

When bringing in new cache lines, an existing entry has to be evicted: **Least Recently Used (LRU)**

- Evict cache line whose last access is longest ago.
- → Least likely to be needed any time soon.

**First In First Out (FIFO)**

- Behaves often similar like LRU.
- But easier to implement.

**Random**

- Pick a random cache line to evict.
- Very simple to implement in hardware.

Replacement has to be decided **in hardware** and **fast**.
What Happens on a Write?

To implement memory writes, CPU makers have two options:

**Write Through**
- Data is directly written to lower-level memory (and to the cache).
  - Writes will stall the CPU.\(^2\)
  - Greatly simplifies data coherency.

**Write Back**
- Data is only written into the cache.
- A *dirty* flag marks modified cache lines (Remember the status field.)
  - May reduce traffic to lower-level memory.
  - Need to write on eviction of dirty cache lines.

Modern processors usually implement **write back**.

\(^2\)Write buffers can be used to overcome this problem.
Putting it all together

To compensate for slow memory, systems use caches.

- Typically multiple levels of caching (memory hierarchy).
- Caches are organized into cache lines.
- **Set associativity**: A memory block can only go into a small number of cache lines (most caches are set-associative).

Systems will benefit from **locality** of data and code.
Performance (SPECint 2000)

- L1 Instruction Cache
- L2 Cache (shared)

Benchmark programs:
- gzip
- vpr
- gcc
- mcf
- crafty
- parser
- eon
- perlbench
- gap
- vortex
- bzip2
- twolf
- avg
Performance (SPECint 2000)

- Benchmark programs: gzip, vpr, gcc, mcf, crafty, parser, eon, perlbench, gap, vortex, bzip2, twolf, avg, TPC-C
- Metrics: Misses per 1000 instructions
- Cache types: L1 Instruction Cache, L2 Cache (shared)
Why do DBSs show such poor cache behavior?

Poor code locality:

- Polymorphic functions
  
  *(E.g., resolve attribute types for each processed tuple individually.)*

- Volcano iterator model (pipelining)
  
  *Each tuple is passed through a query plan composed of many operators.*
Why do DBSs show such poor cache behavior?

Poor data locality:

- Database systems are designed to navigate through large data volumes quickly.
- Navigating an index tree, e.g., by design means to “randomly” visit any of the (many) child nodes.
References I

What Every Programmer Should Know About Memory.

OLTP through the looking glass, and what we found there.
In *SIGMOD*, pages 981–992.

*Computer Architecture: A Quantitative Approach*.
Morgan Kaufmann, 2 edition.

*Computer Architecture: A Quantitative Approach*.
Morgan Kaufmann, 4 edition.

*Computer Architecture - A Quantitative Approach*.
Morgan Kaufmann, 5 edition.

The end of an architectural era: (it’s time for a complete rewrite).
In *VLDB*, pages 1150–1160.
Web Resources


