Main-Memory

Database Management Systems

Sebastian Dorok
Bayer Pharma AG and Otto-von-Guericke University Magdeburg
Summer Term 2015
We will talk about

The Past and the Present

Cache Awareness

Processing Models

Storage Models

Design of Main-Memory DBMSs for Database Applications

Summary
Motivation: The Past and the Present
The Past - Computer Architecture

<table>
<thead>
<tr>
<th>Latency</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 ns</td>
<td>200 B</td>
</tr>
<tr>
<td>10 ns</td>
<td>64 KB</td>
</tr>
<tr>
<td>100 ns</td>
<td>32 MB</td>
</tr>
<tr>
<td>5.000.000 ns</td>
<td>2 GB</td>
</tr>
</tbody>
</table>

Data taken from [Hennessy and Patterson, 1996]
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...
The Present - Computer Architecture

latency capacity
300 ps 1000 B
1 ns 64 kB
3 - 10 ns 256 kB
10 - 20 ns 2 - 4 MB
50 - 100 ns 4 - 16 GB
5.000.000 - 10.000.000 ns 4 - 16 TB

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

- Hundreds to thousands of gigabyte of main memory available
  - Up to $10^6$ times more capacity!
  - Complete database having less than a TB size can be kept in main memory
  - Use main memory as primary storage for the database and remove disk access as main performance bottleneck
The Present - Database Systems

- Hundreds to thousands of gigabyte of main memory available
  → Up to $10^6$ times more capacity!
  → Complete database having less than a TB size can be kept in main memory
  → 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

- CPU
- Main Memory
  - Buffered Data
- Disk
  - Data

Main-Memory DBMS

- CPU
- Main Memory
  - Data
- Disk
  - Replicated Data
ATTENTION: Main-memory storage ≠ No Durability

→ ACID properties have to be guaranteed

→ However, there are new ways of guaranteeing it, such as a second machine in hot standby
Disk-based vs. Main-Memory DBMS (3)

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

→ Remove level of indirection

→ Results in better performance
Disk-based vs. Main-Memory DBMS (4)

Disk bottleneck is removed as database is kept in main memory

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

Accessing main-memory is much more expensive than accessing CPU registers.

→ Is main-memory the new disk?
Rethink the Architecture of DBMSs

Even if the complete database fits in main memory, there are significant overheads of traditional, System R like DBMSs:

- Many function calls $\rightarrow$ stack manipulation overhead$^1$ + instruction-cache misses
- Adverse memory access $\rightarrow$ data-cache misses

$\rightarrow$ Be aware of the caches!

---

$^1$Can be reduced by function inlining
Cache Awareness

Partially based on *Processing on Modern Hardware* by Jens Teubner from TU Dortmund.
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)
Hardware Trends

normalized performance

year


Processor

DRAM Memory

Data taken from [Hennessy and Patterson, 2012]
Hardware Trends

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**
  - SRAM: bytes, latency < 1 ns

- **L1 Cache**
  - SRAM: kilobytes, latency ≈ 1 ns

- **L2 Cache**
  - SRAM: megabytes, latency < 10 ns

- **main memory**
  - DRAM: gigabytes, latency 70–100 ns

- **disk**

- Some systems also use a 3rd level cache.
- cf. Architecture & Implementation course
  - Caches resemble the buffer manager but are **controlled by hardware**
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.
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.\(^2\)

\(^2\)Modern 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 ($\approx 1$ ns)
- **L2 hit latency**: 7 cycles ($\approx 3.5$ ns)
- **L2 miss latency**: 160–180 cycles ($\approx 60$ 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
  \[ \rightarrow \text{4 MB cache, line size: 64 B: 65,536 cache lines.} \]
- Used, *e.g.*, for small TLB caches.

---

Sebastian Dorok  
Main-Memory Database Management Systems  
Last Change: May 11, 2015  
23/116
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 showing block placement in a direct-mapped cache]

place block 12 in cache line 4 
\(4 = 12 \mod 8\)
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]

- 512 kB
- 1 MB
- 2 MB
- 4 MB
- 8 MB
- 16 MB

- 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**.

![Diagram showing the structure of a cache line with status, tag, and data fields, and how the tag can be derived from the byte address.](image)
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 \((4 \text{ MB} ÷ 64 \text{ bytes})\).
- **Associativity:** 16-way set-associative.
  - → There are 4,096 sets \((65,536 ÷ 16 = 4,096)\).
  - → 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 \((36 - 12 - 6 = 18)\).
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.  
  → 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.  

3Write buffers can be used to overcome this problem.
Putting it all Together

To compensate for slow memory, systems use caches.

- DRAM provides high capacity, but long latency.
- SRAM has better latency, but low capacity.
- 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)

```
0  5  10  15  20
misses per 1000 instructions
```

- L1 Instruction Cache
- L2 Cache (shared)

```
gzip  vpr  gcc  mcf  crafty  parser  eon  perlbench  gap  vortex  bzip2  twolf  avg
benchmark program
```
Performance (SPECint 2000)

![Graph showing misses per 1000 instructions for various benchmark programs]

- **L1 Instruction Cache**
- **L2 Cache (shared)**

**Benchmark Programs:**
- gzip
- vpr
- gcc
- mcf
- crafty
- parser
- eon
- perlbench
- gap
- vortex
- bzip2
- twolf
- avg
- TPC-C
Why do DBSs show such poor cache behavior?

Poor code locality:

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

  \[
  \text{TYPE}
  \]
  \[
  \text{tupleGetNthAttribute(N)}
  \]
  \[
  \begin{align*}
  &\text{String} \\
  &\text{tupleGetNthAttribute(N)}
  \end{align*}
  \]
  \[
  \begin{align*}
  &\text{Int} \\
  &\text{tupleGetNthAttribute(N)}
  \end{align*}
  \]
  \[
  \begin{align*}
  &\text{Float} \\
  &\text{tupleGetNthAttribute(N)}
  \end{align*}
  \]
  \[
  \begin{align*}
  &\text{Double} \\
  &\text{tupleGetNthAttribute(N)}
  \end{align*}
  \]

• 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.