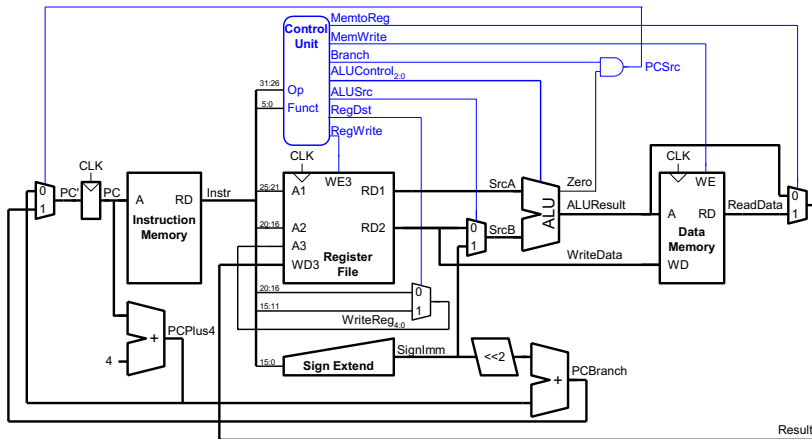


# Performance: Memory Hierarchy, Caches

Becker/Molitor, Chapter 11.4

Jan Reineke  
Universität des Saarlandes

# Roadmap: Computer architecture



1. Combinatorial circuits: Boolean Algebra/Functions/Expressions/Synthesis
2. Number representations
3. Arithmetic Circuits: Addition, Multiplication, Division, ALU
4. Sequential circuits: Flip-Flops, Registers, SRAM, Moore and Mealy automata
5. Verilog
6. Instruction Set Architecture
7. Microarchitecture
8. **Performance:** RISC vs. CISC, Pipelining, Memory Hierarchy

# The perfect memory

On-chip memory that is

*fast*, = low access latency

*large*, = high capacity

*cheap* = low area consumption

# Key insight:

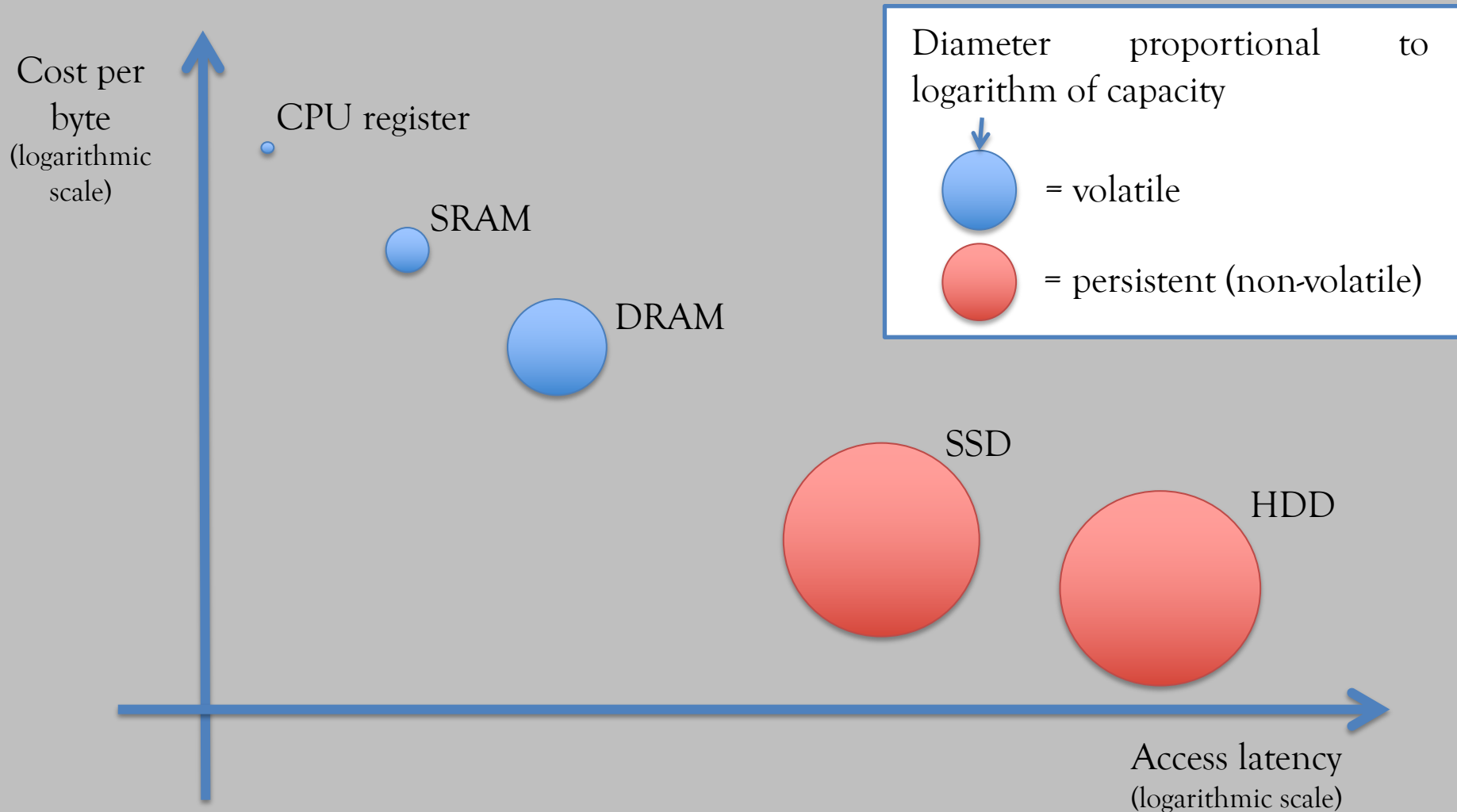
## A perfect memory is impossible

There are tradeoffs between *capacity*, *cost* and *access latency*:

1. A larger memory capacity implies a greater area consumption (= higher cost).
2. A greater area consumption implies greater distances between different memory cells.
3. Greater distances imply greater propagation delays and thus higher access latencies.

→ Larger memories have higher access latencies.

# Differences between memory technologies



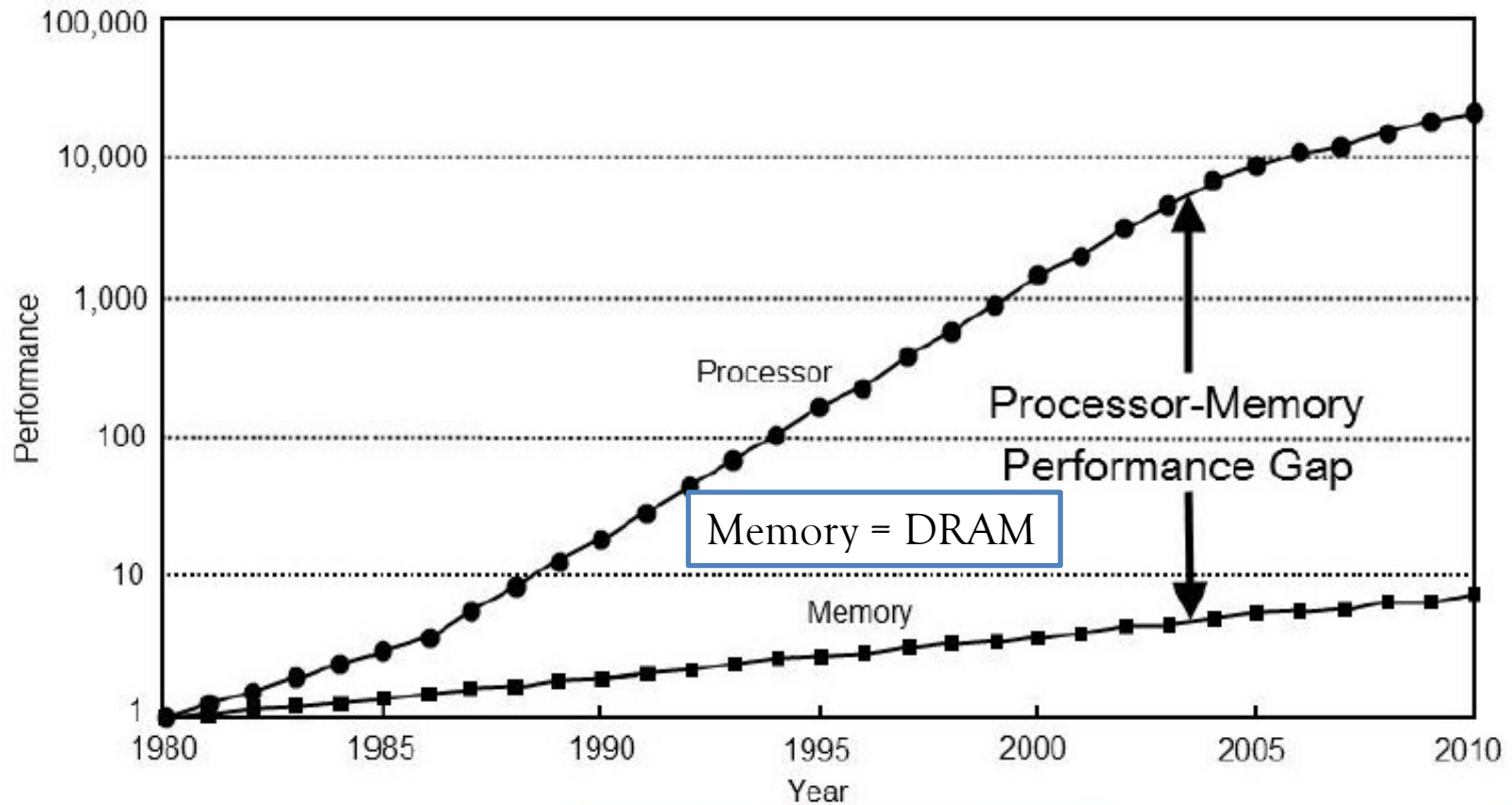
# Historical review

*“Ideally, one would desire an  
indefinitely large memory capacity  
such that any particular [...] word would be  
immediately available.*

*We are [...] forced to recognize the possibility of constructing  
a **hierarchy of memories**, each of which has greater  
capacity than the preceding but which is less quickly  
accessible.”*

A. W. Burke, H. H. Goldstine, and J. von Neumann (1946)

# Processor vs DRAM: Memory Gap



# Memory hierarchy

*Goal:* Illusion of a  
very fast memory (access latency  $\approx 1$  processor cycle), with  
very high capacity (several TBs).

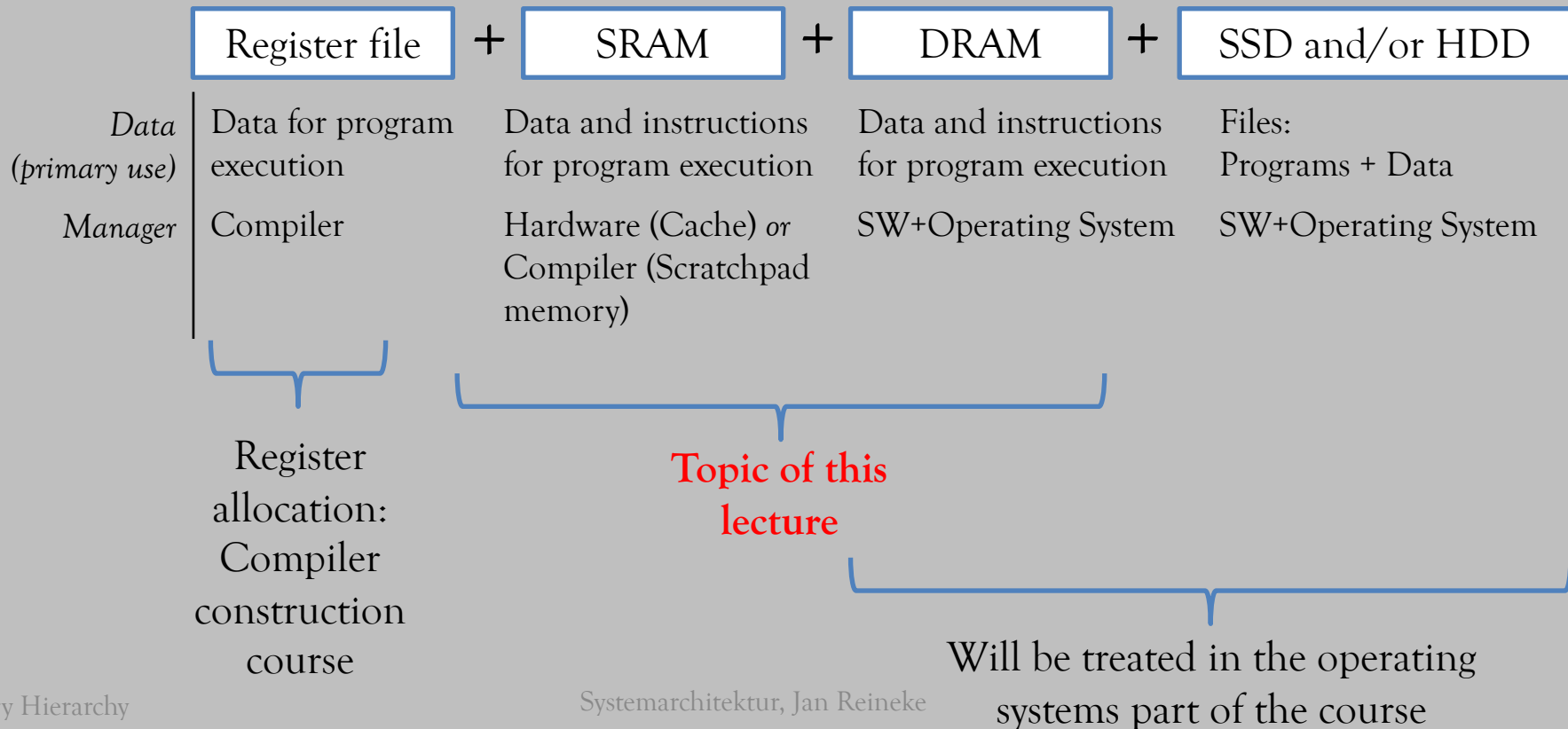
Is achieved by a clever combination of *smaller, faster, and larger, slower* memories:

	Register file	+	SRAM	+	DRAM	+	SSD and/or HDD
Capacity	$\approx 1$ KB		32 KB (L1 Cache) – 16384 KB (L3 Cache)		$\approx 16$ GB		$\approx 1$ TB
Access latency	$< 1$ cycle		1-3 cycles (L1 Cache) – $\approx 40$ cycles (L3 Cache)		100 – 400 cycles		3 – 12 ms (HDD) $\approx 10^7$ cycles $< 0.1$ ms (SSD) $\approx 10^5$ cycles
Throughput	no bottleneck		no bottleneck		$\approx 51$ GB/s (DDR5) $\approx 665$ GB/s (HBM3)		$\approx 200$ MB/s (HDD) $\approx 7000$ MB/s (SSD)



# Memory Hierarchy

- Which data is stored in the register file, SRAM, DRAM, SSD/HDD?
- Who makes the decision?



# SRAM to accelerate memory accesses:

## Option I: **Scratchpad Memory**

A small part of the address space is serviced via SRAM, the rest via DRAM:

Address space:



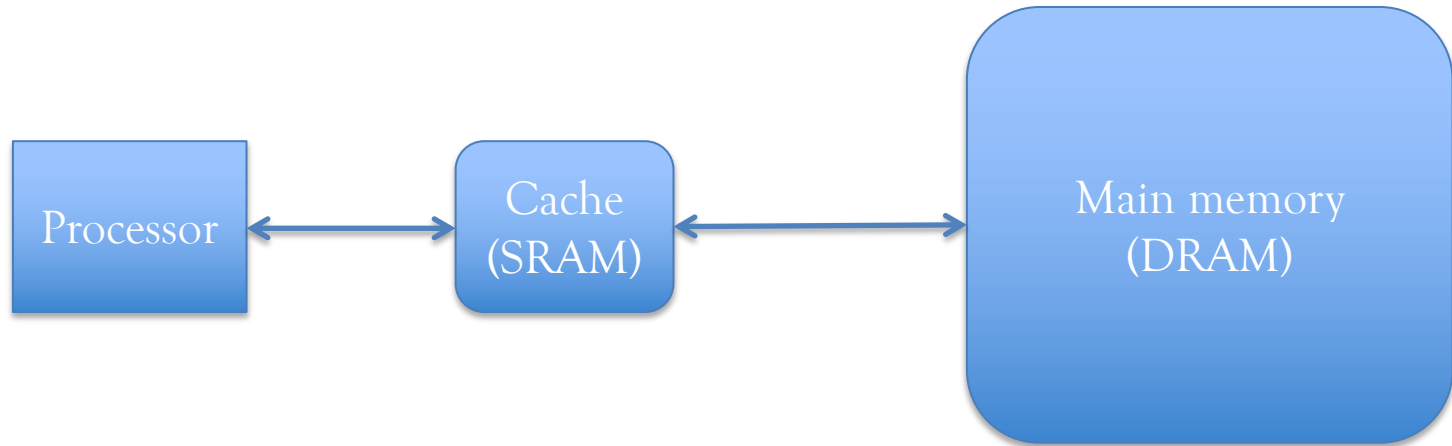
Division of data between SRAM and DRAM determined by software, as it decides where to place which data.

- Mostly used in embedded systems, seldom in PCs.  
Was used in the Cell processor (PlayStation 3).
- Unpopular, as it requires software adaptation.

# SRAM to accelerate memory accesses :

## Option II: **Caches**

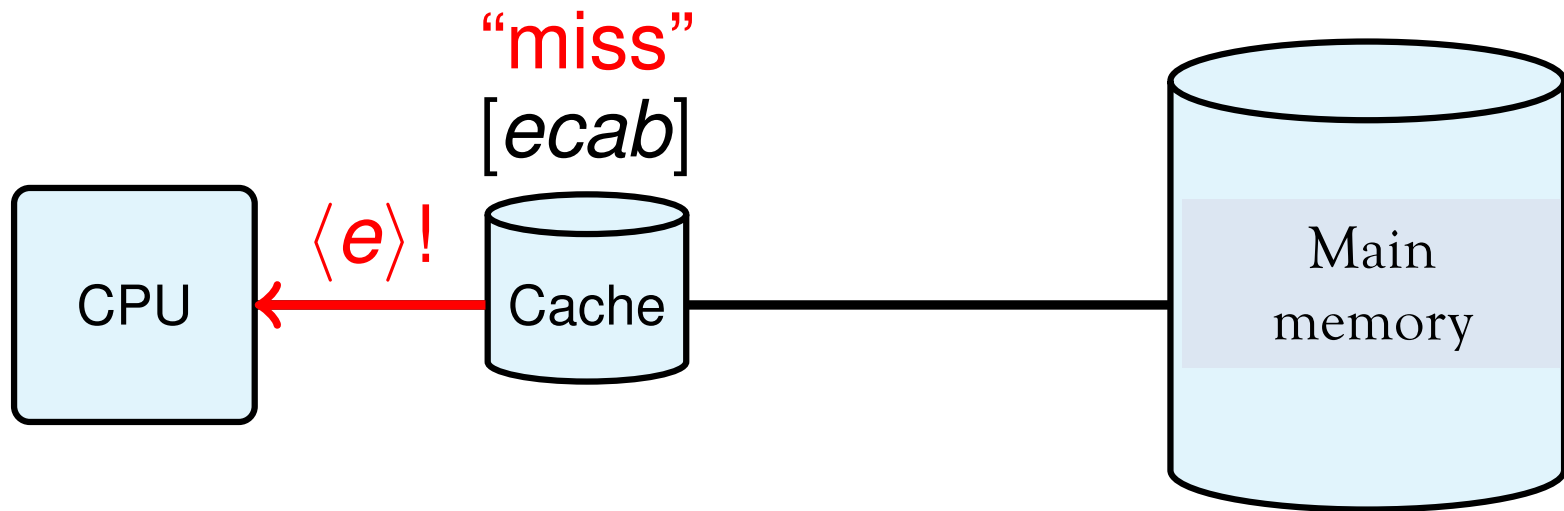
**Caches** store a **subset** of the data from main memory (DRAM):



Contents of the cache are **determined dynamically** by hardware, based on the memory accesses.

- Used in PCs, but also in many embedded systems.
- Popular, as it is **transparent** to software.
  - No change of software required to use the cache.

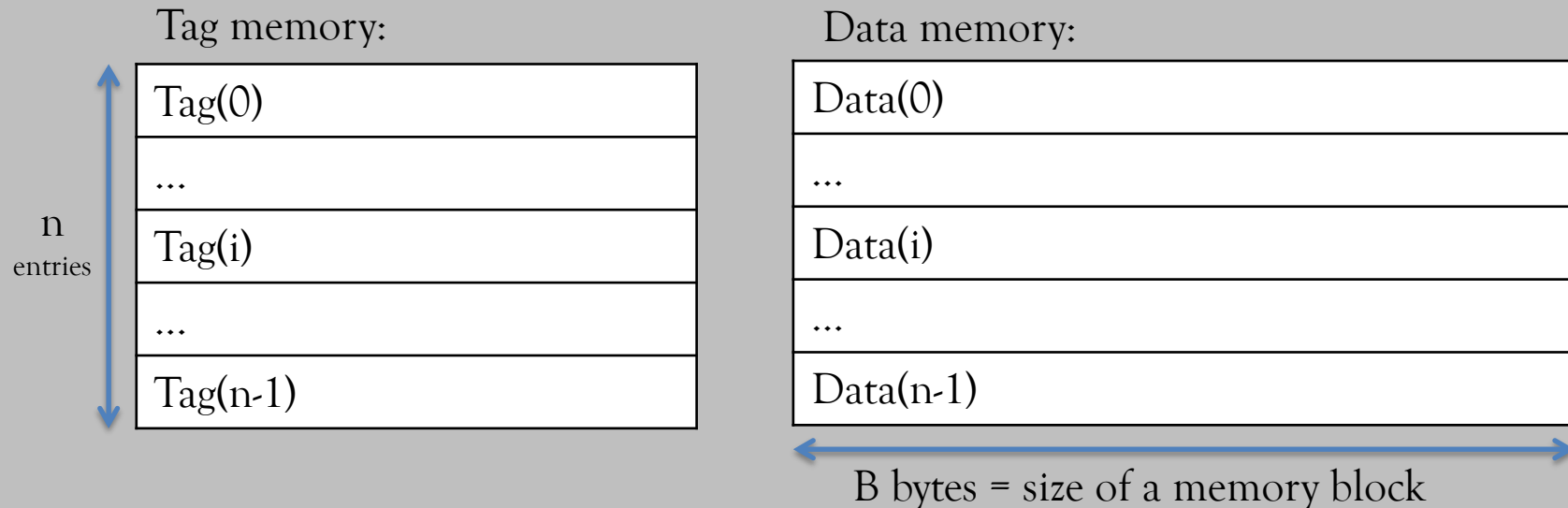
# Caches: High-level behaviour



# Cache organization

It is *not sufficient*, to simply store data in the cache:  
the cache needs to remember **which addresses** are cached.

→ Cache is divided into *tag* and *data* memory:



# Cache implementation questions

1. Where in the cache are memory blocks located?  
How are they retrieved?  
→ *fully-associative, direct-mapped, set-associative*
2. Which block is replaced upon a cache miss?  
→ *replacement policy*
3. How large are the memory blocks stored in cache?

# Where in the cache are memory blocks located?

## Fully-associative cache:

Every memory block can be stored in **any** locations of the data memory of the cache

## Direct-mapped cache:

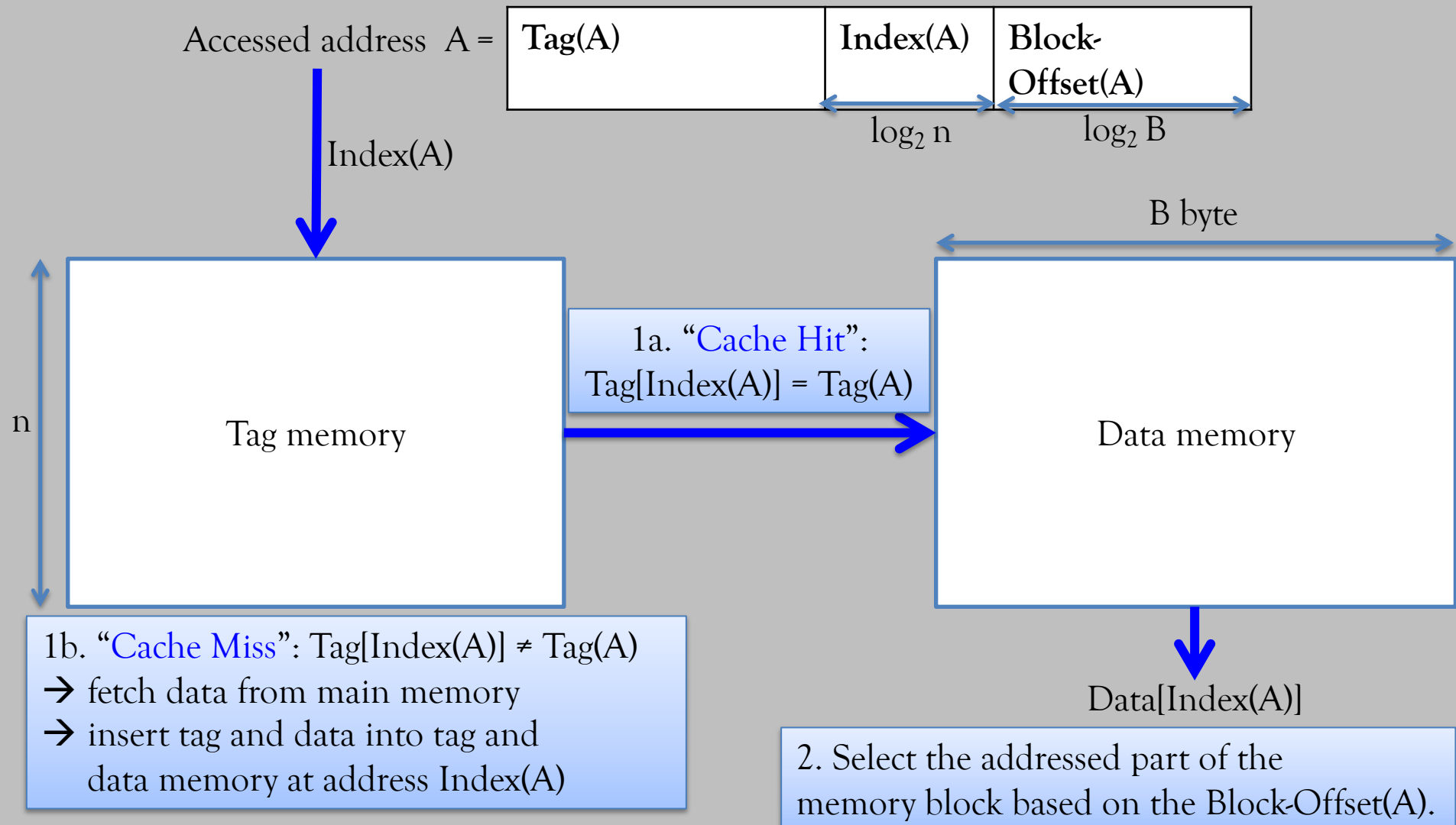
Every memory block can be stored in **exactly one** locations of the data memory of the cache

## Set-associative cache:

Compromise between the two extreme cases: every memory block can be stored in a **fixed subset** of all locations in the data memory

*Example:*

# Cache access for a direct-mapped cache





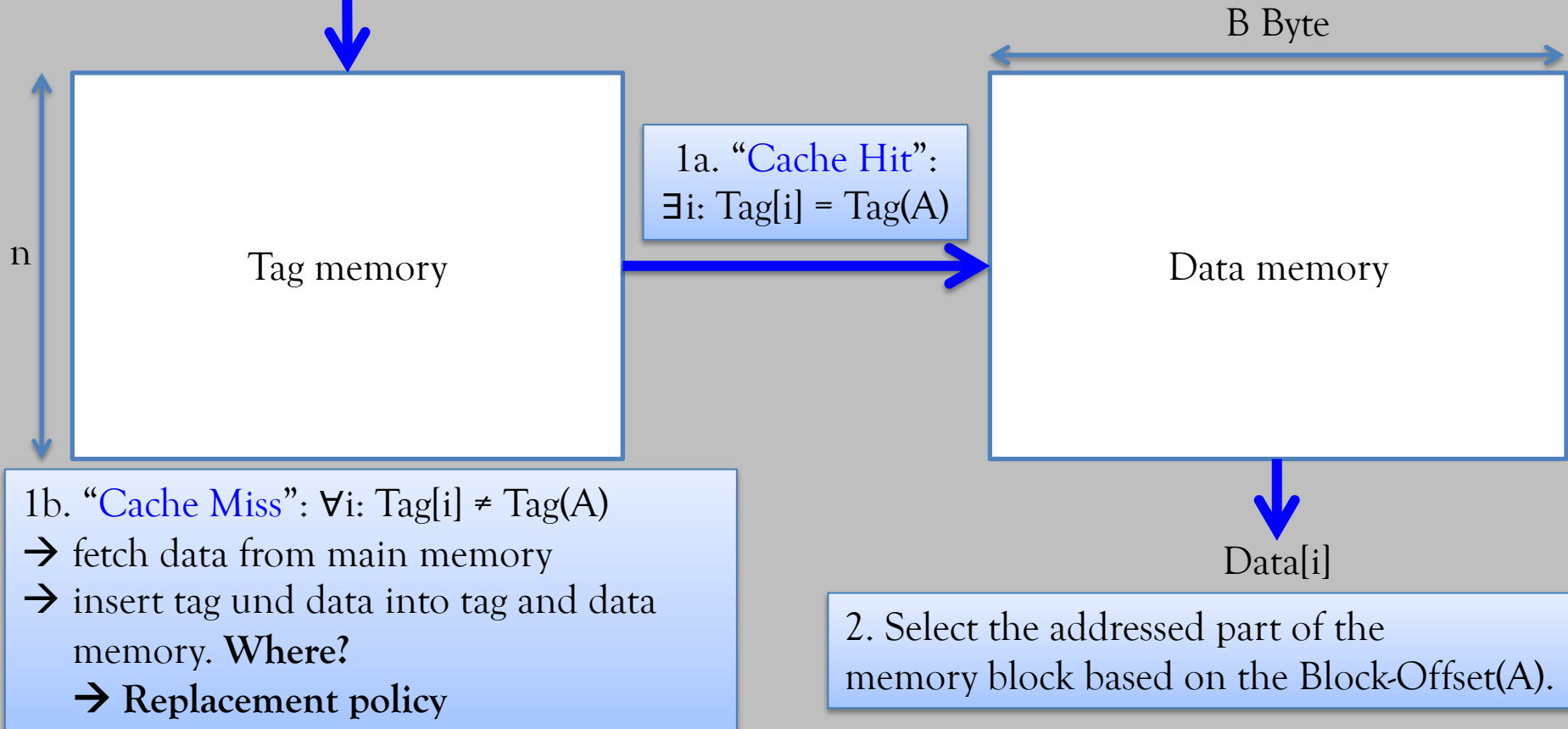
*Example:*

# Cache access for a fully-associative cache

Accessed address  $A =$ 

$\text{Tag}(A)$	$\text{Block-Offset}(A)$
-----------------	--------------------------

  
 $\log_2 B$



# Direct-mapped vs fully-associative caches

	fully-associative	direct-mapped
Location of data	freely chosen	fully determined by index-mapping
Localizing the data	parallel comparison with all tags	<b>single comparison</b> with tag at Index(A)
Replacement policy	yes	not necessary

- fully-associative caches requires expensive parallel comparison of tags  
(or slow serial comparison of tags)
- as a consequence, only very small fully-associative caches exist in practice

**Set-associative caches** are a compromise:

The index of an address determines a small set of locations, whose tags are then compared in parallel as in the fully-associative cache.

## 2. Which memory is replaced upon a cache miss?

The **replacement policy** (also eviction policy) determines which memory block to replace upon a miss.

*Goal:* Minimizing the number of cache misses.

Difficult, as only past memory accesses are known to the cache.

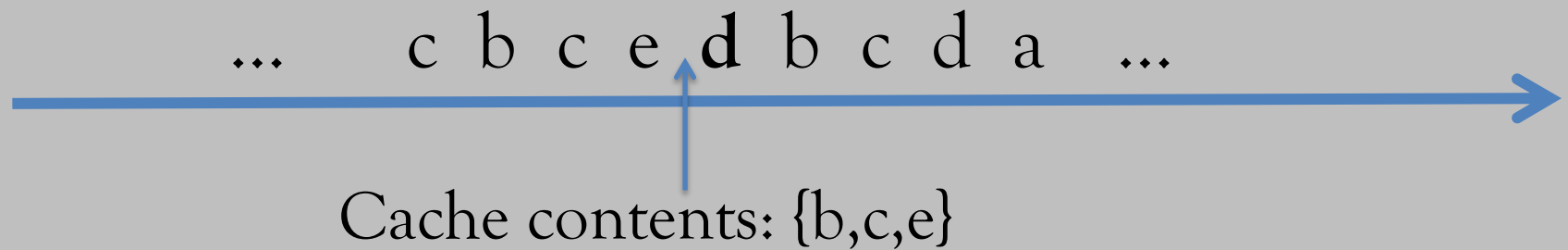
*Prediction is very difficult,  
especially if it's about the future.*



Niels Bohr (physicist)

# Gedankenexperiment: An optimal replacement policy

A memory-access sequence:



Which block should be replaced upon the access to **d**?

**b, c or e?**

→ In this situation, **e** should be replaced,  
as **b** and **c** are required again earlier.

# Gedankenexperiment: An optimal replacement policy

## Farthest-in-the-Future (OPT):

Replace the block whose next access is farthest in the future.

## Theorem

OPT minimizes the number of cache misses.

**Problem:** OPT cannot be implemented, as it requires knowledge about future memory accesses.

→ OPT is a so-called *offline algorithm*

→ Practically realizable algorithms are *online algorithms*,  
they can only rely on past accesses in their decision making

## 2. Which block is replaced upon a cache miss?

Popular online replacement policies:

**Least-recently-used (LRU):**

Replace the block that has been used least recently.

“the best predictor of the future... is the past”

**First in, first out (FIFO):**

Replace the oldest block in the cache.

→ Cheaper to implement in hardware than LRU.

# How to evaluate replacement policies?

- *Empirical analysis:*  
Comparison of the number of cache misses on benchmark programs.
- *Theoretical analysis:*  
Comparison with OPT.

Let  $\text{LRU}_k(s)$  be the number of cache misses of LRU on a fully-associative cache of size  $k$  on the access sequence  $s$ .

Let  $\text{FIFO}_k(s)$  and  $\text{OPT}_k(s)$  be defined analogously.

## **Theorem (LRU vs OPT, FIFO vs OPT):**

Let  $s$  be an arbitrary access sequence and  $k > 0$ . Then:

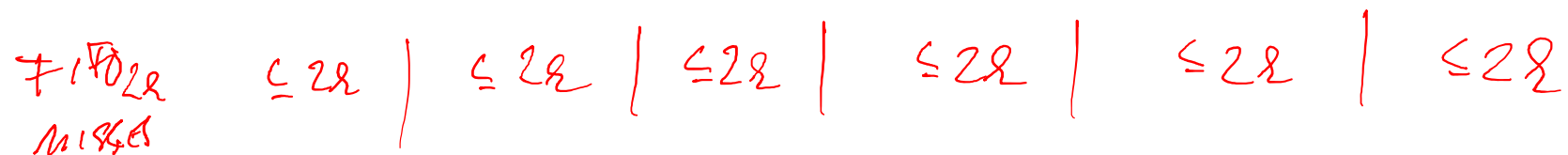
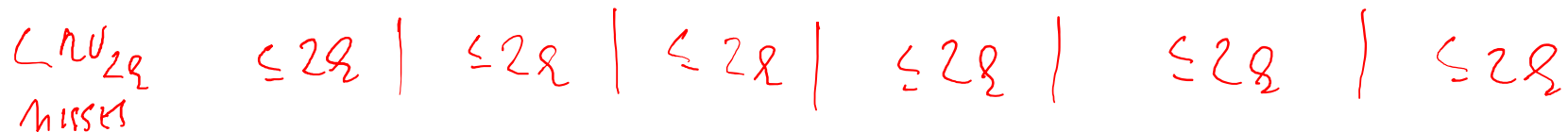
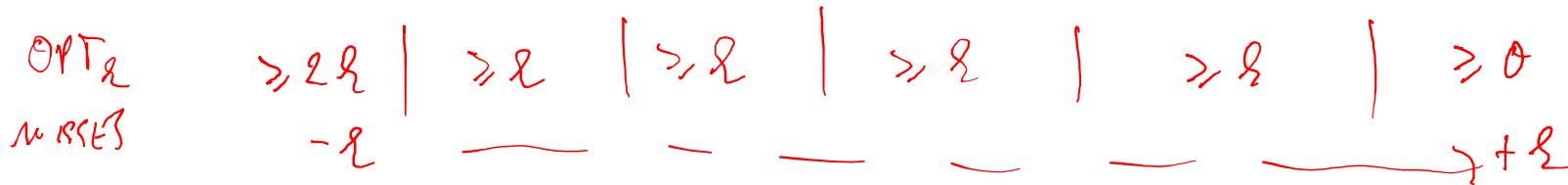
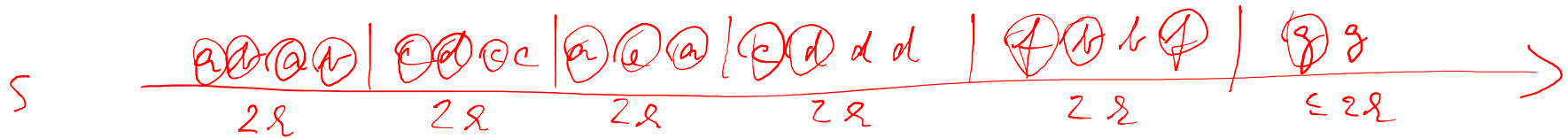
$$\text{LRU}_{2k}(s) \leq 2 \cdot \text{OPT}_k(s) \quad \text{and} \quad \text{FIFO}_{2k}(s) \leq 2 \cdot \text{OPT}_k(s)$$

## Theorem (LRU vs OPT, FIFO vs OPT):

Let  $s$  be an arbitrary access sequence and  $k > 0$ . Then:

$$\text{LRU}_{2k}(s) \leq 2 \cdot \text{OPT}_k(s) \quad \text{and} \quad \text{FIFO}_{2k}(s) \leq 2 \cdot \text{OPT}_k(s)$$

EXAMPLE  $k=7$





# Why is the memory hierarchy effective in practice?

## Principle of locality

- **Temporal locality:**  
After accessing address  $x$ , the same address  $x$  is often accessed again soon after.
- **Spatial locality:**  
After accessing address  $x$ , its neighboring addresses are often accessed next.

### *Examples:*

*Instruction fetches:* Loops in a program, recursive functions

*Data:* Divide-and-conquer algorithms

# Sizes of memory blocks

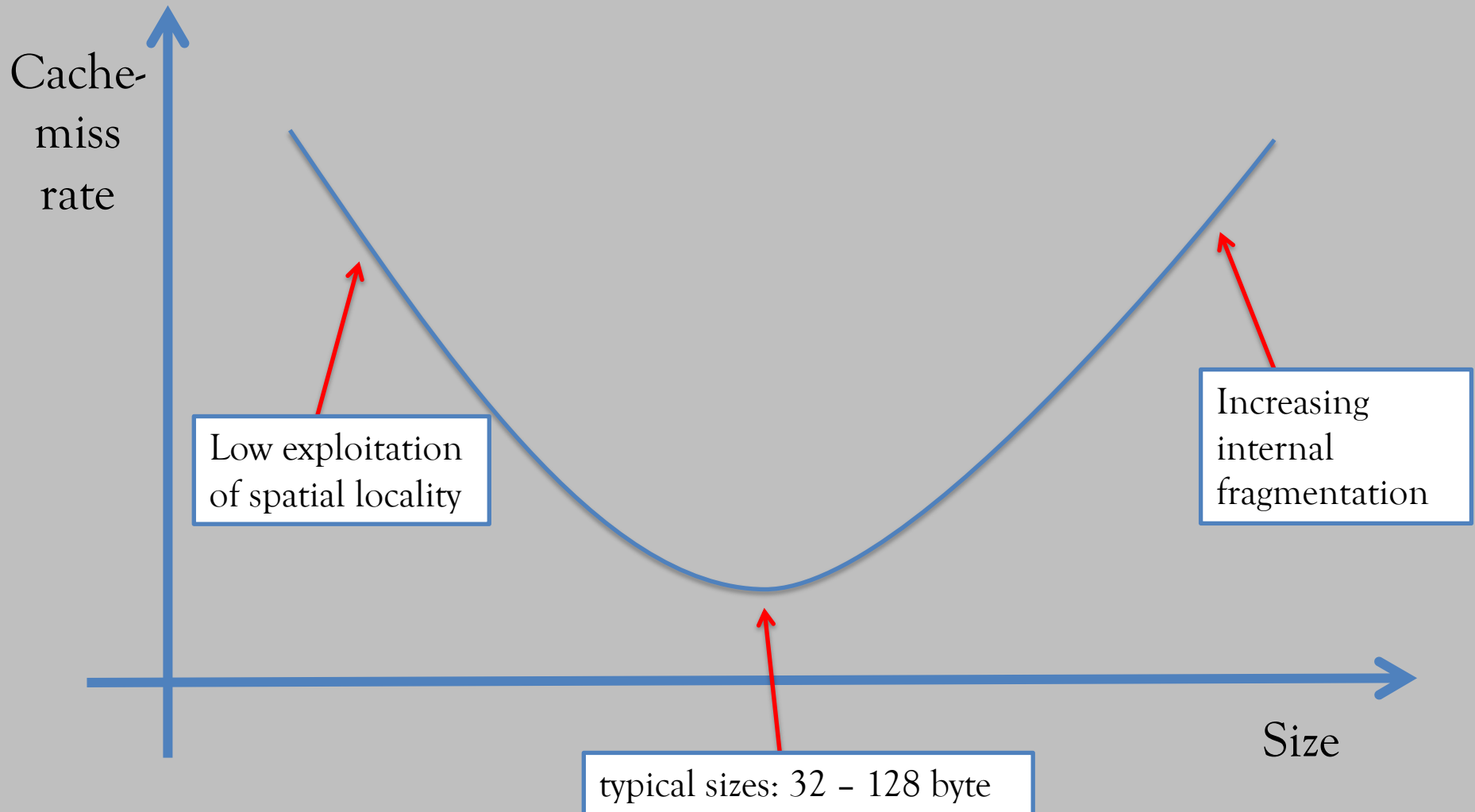
## *Advantages of large memory blocks:*

- greater exploitation of spatial locality
- less overhead due to meta data (e.g. tag memory)

## *Disadvantages of large memory blocks:*

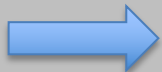
- higher “miss penalty”  
(= cost of fetching a memory block from main memory into the cache)
- Potentially waste of cache capacity  
(internal fragmentation)

# Cache-miss rate in terms of size of memory blocks (for fixed cache size)



# Performance: Influence of the cache

- Terminology:
  - **Hit rate** = Share of all memory accesses that are hits
  - **Miss rate** =  $1 - \text{Hit rate}$
  - **Miss penalty** = Additional memory latency upon a miss
- $\text{Execution time} = (\text{CPU cycles} + \text{Memory stall cycles}) \cdot \text{Cycle time}$   
where  $\text{CPU cycles} = \text{Number of instruction} \cdot \text{CPI}_{\text{hit}}$   
and  $\text{Memory stall cycles} = \text{Number of misses} \cdot \text{Miss penalty}$ 
$$= \text{Number of instructions} \cdot \frac{\text{Misses}}{\text{Instruction}} \cdot \text{Miss penalty}$$
$$= \text{Number of instructions} \cdot \frac{\text{Memory accesses}}{\text{Instruction}} \cdot \text{Miss rate} \cdot \text{Miss penalty}$$



*Execution time*

$$= \text{Number of instructions} \cdot \left( \text{CPI}_{\text{hit}} + \frac{\text{Memory accesses}}{\text{Instruction}} \cdot \text{Miss rate} \cdot \text{Miss penalty} \right) \cdot \text{Cycle time}$$

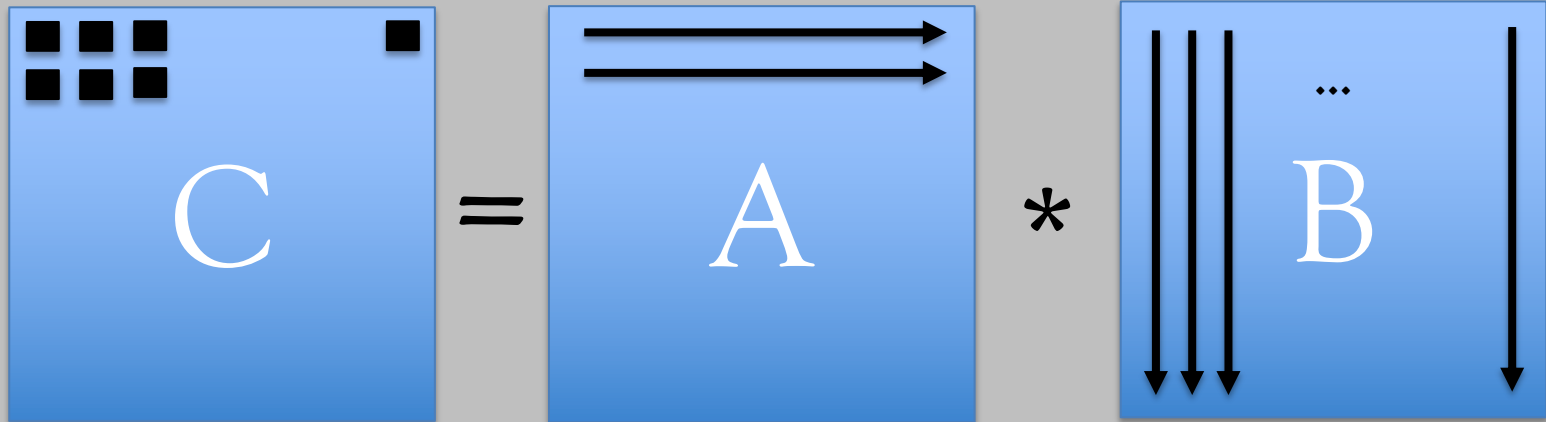
# Performance: Influence of the cache

- Typical values:
  - $CPI_{hit} = 2$
  - Miss penalty = 100
  - Memory accesses/instruction = 1,2
- Plugging those values in yields:  
*Execution time*  
$$= \text{Number of instructions} \cdot \underbrace{(2 + 1,2 \cdot 100 \cdot \text{Miss rate})}_{CPI} \cdot \text{Cycle time}$$
- Effect of different miss rates:
  - Miss rate = 1       $\rightarrow CPI = 122$
  - Miss rate = 0,1     $\rightarrow CPI = 14$
  - Miss rate = 0,01    $\rightarrow CPI = 3,2$
  - Miss rate = 0       $\rightarrow CPI = 2$

$\rightarrow$  Need very low miss rates!

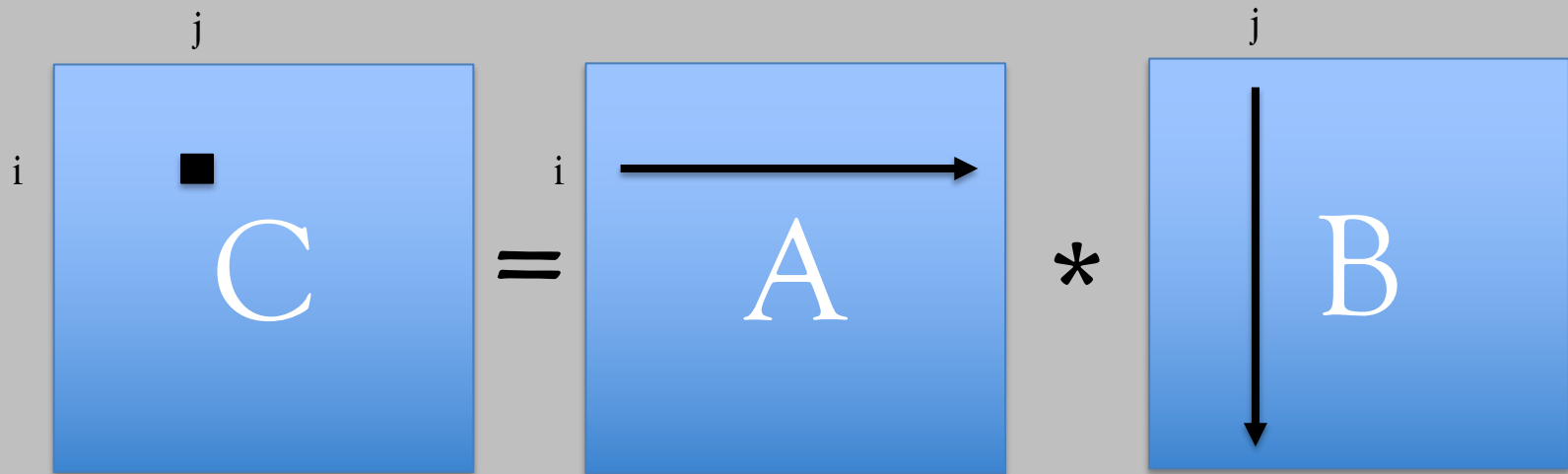
# Example: Matrix multiplication

```
Variant 1: for (int i = 0; i < n; i++)  
            for (int j = 0; j < n; j++)  
                for (int k = 0; k < n; k++)  
                    C[i][j] += A[i][k] * B[k][j];
```



# Work analysis: Computations

```
Variant 1: for (int i = 0; i < n; i++)  
            for (int j = 0; j < n; j++)  
                for (int k = 0; k < n; k++)  
                    C[i][j] += A[i][k] * B[k][j];
```



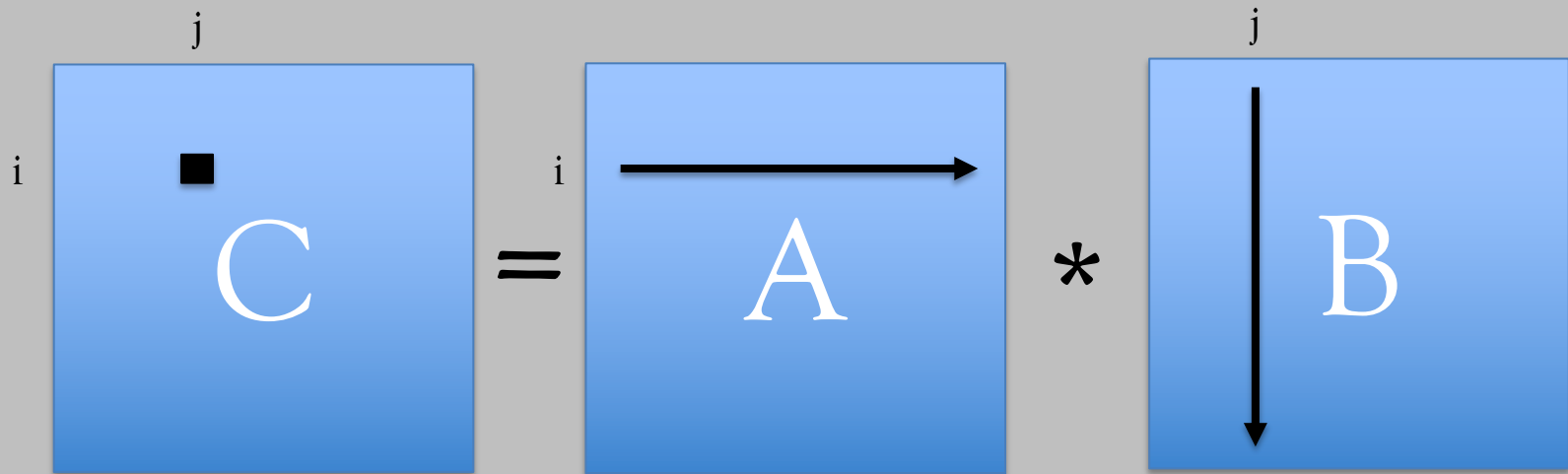
Work analysis: Number of multiplications/additions per inner loop?

→  $n$  multiplications,  $n$  additions

→ In total  $\Theta(n^3)$  operations

# Work analysis: Loads

```
Variant 1: for (int i = 0; i < n; i++)  
            for (int j = 0; j < n; j++)  
                for (int k = 0; k < n; k++)  
                    C[i][j] += A[i][k] * B[k][j];
```



Work analysis: Number of loads per inner loop?

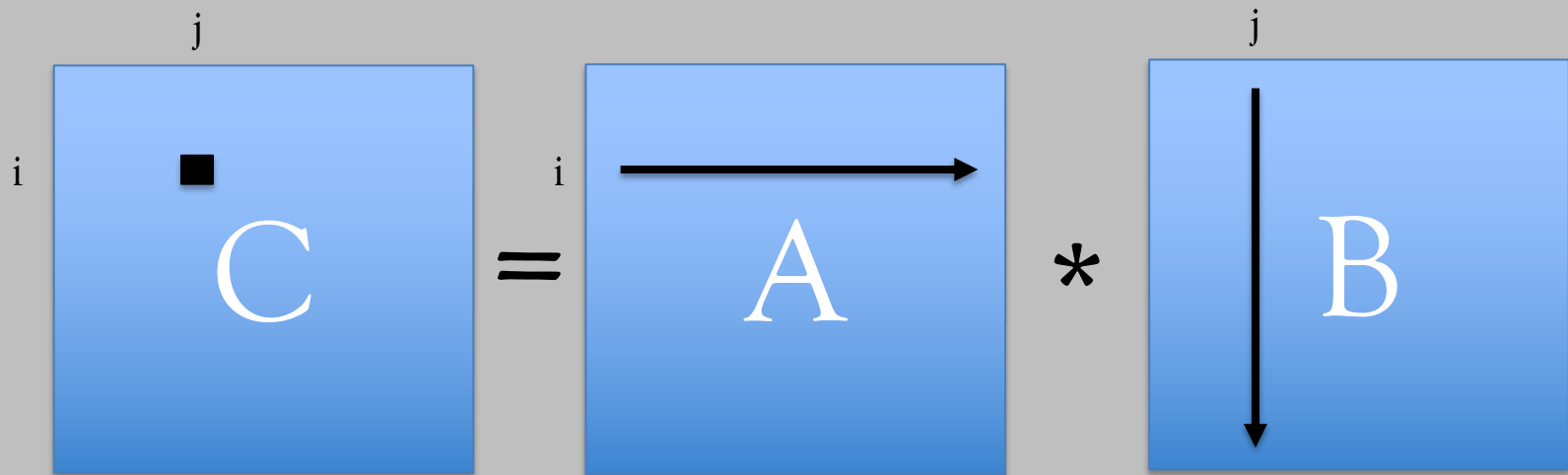
→  $3n$  loads

→ In total  $3n^3$  loads



# Work analysis: Cache misses

```
Variant 1: for (int i = 0; i < n; i++)  
            for (int j = 0; j < n; j++)  
                for (int k = 0; k < n; k++)  
                    C[i][j] += A[i][k] * B[k][j];
```



Work analysis: Total number of cache misses?

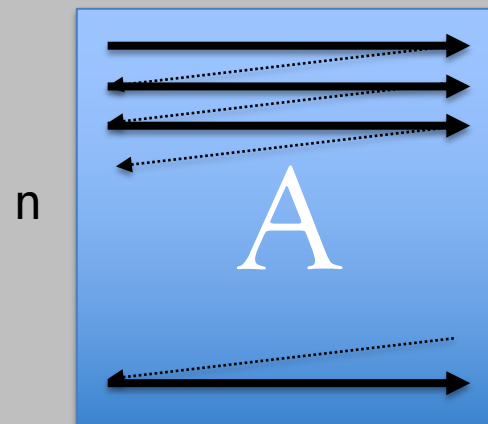
# Memory layout of arrays

How are two-dimensional arrays stored in memory?

→ typically **row-major layout** (alternative: column-major layout)

Example:

```
int matrixA[n][m]
```

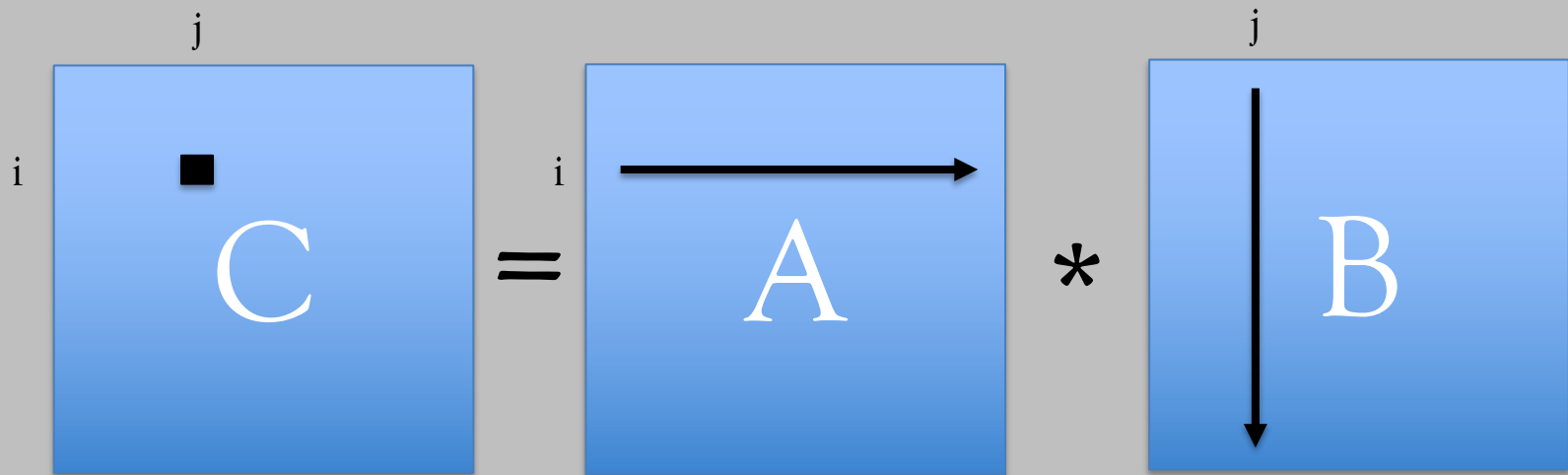


Memory:	$A[0][0]$	$A[0][1]$	$A[0][2]$	...	$A[0][m-1]$	$A[1][0]$	$A[1][1]$	$A[1][2]$	...	$A[1][m-1]$	...	$A[n-1][m-1]$
Address:	$A$	$A+4$	$A+8$	...	$A+4m$	...						$A+4nm-4$

Assuming memory blocks of size  $B=32$  bytes, we have 8 consecutive entries per block.

# Work analysis: Cache misses

```
Variant 1: for (int i = 0; i < n; i++)  
            for (int j = 0; j < n; j++)  
                for (int k = 0; k < n; k++)  
                    C[i][j] += A[i][k] * B[k][j];
```



Work analysis: Cache misses in the first execution of inner loop?

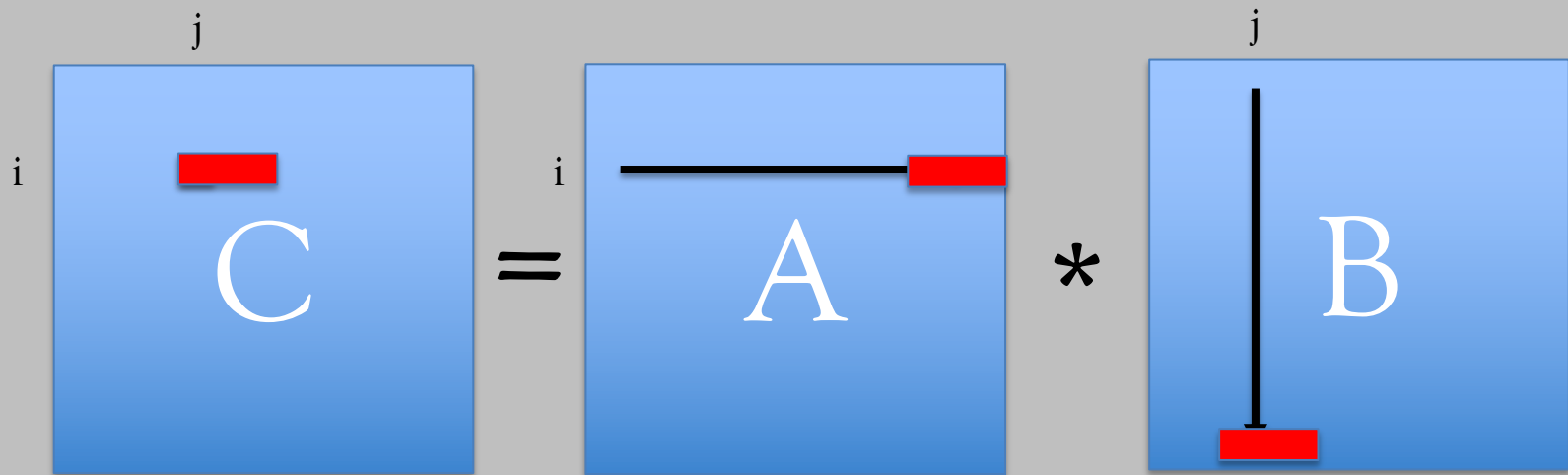
1 miss to C

$4n/B$  misses to A

n misses to B

# Work analysis: Cache misses

```
Variant 1: for (int i = 0; i < n; i++)  
            for (int j = 0; j < n; j++)  
                for (int k = 0; k < n; k++)  
                    C[i][j] += A[i][k] * B[k][j];
```



Work analysis: Cache misses in total?

Assuming fully-associative cache of size 3 memory blocks  $\ll n$ .

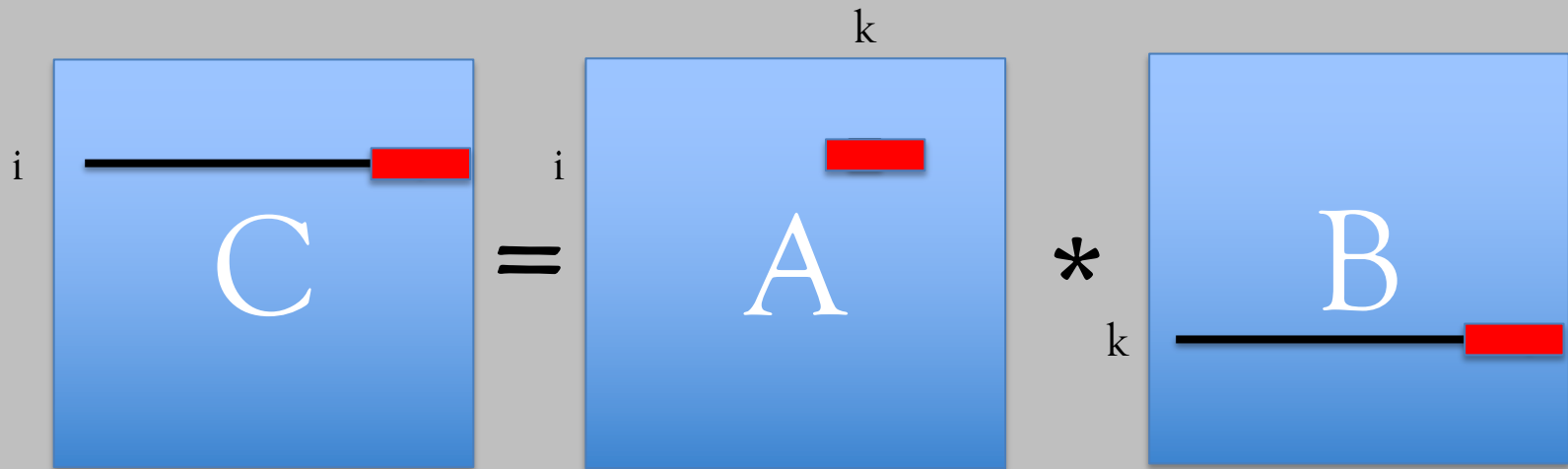
$4n^2/B$  misses to C

$4n^3/B$  misses to A

$4n^3$  misses to B

# Work analysis: Cache misses

Variant 2: `for (int i = 0; i < n; i++)`  
    `for (int k = 0; k < n; k++)`  
        `for (int j = 0; j < n; j++)`  
            `C[i][j] += A[i][k] * B[k][j];`



Work analysis: Cache misses in total?

Assuming fully-associative cache of size 3 memory blocks  $\ll n$ .

$4n^3/B$  misses to C

$4n^2/B$  misses to A

$4n^3/B$  misses to B

# Example: Matrix multiplication

Assumed **very small cache** so far

→ could only exploit spatial locality

Results would be different for **large cache** that fits matrices A, B, C entirely →  $3 \cdot 4n^2/B$  misses in total.

What if cache size is in between the two extremes (realistic!)?

→ Adapt algorithm to increase temporal locality

# Matrix multiplication, Tiling

→ Adapt algorithm to increase temporal locality

Variant 1: 

```
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        for (int k = 0; k < n; k++)  
            C[i][j] += A[i][k] * B[k][j];
```



Tiling

```
for (int i0 = 0; i0 < n; i0+=s)  
    for (int j0 = 0; j0 < n; j0+=s)  
        for (int k0 = 0; k0 < n; k0+=s)  
            for (int i = i0; i < i0+s; i++)  
                for (int j = j0; j < j0+s; j++)  
                    for (int k = k0; k < k0+s; k++)  
                        C[i][j] += A[i][k] * B[k][j];
```

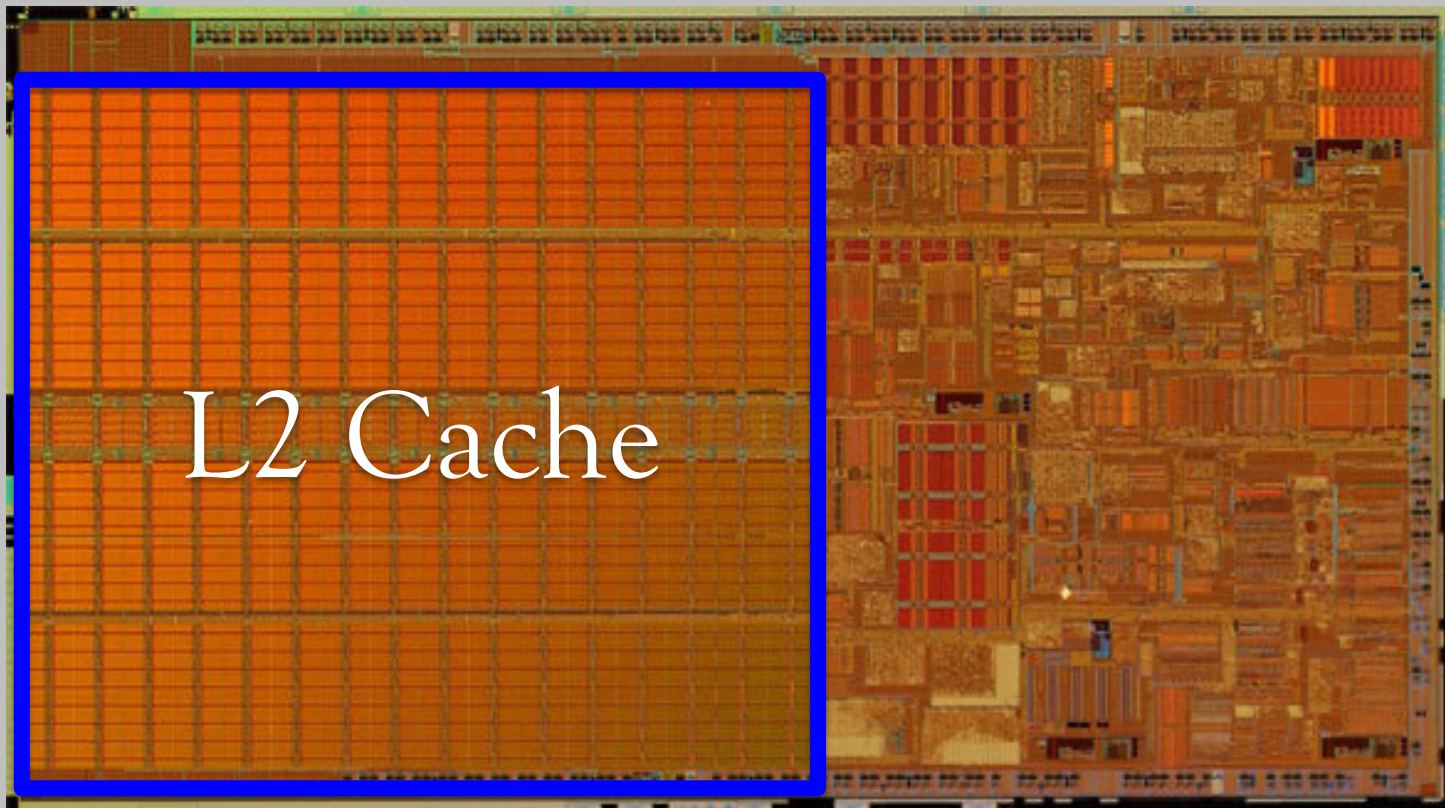
# Summary

- Memory hierarchy:  
Combination of **smaller**, **faster** and  
**larger**, **slower** memories
- Memory hierarchies usually work well due to
  - **spatial locality** and
  - **temporal locality**.
- **Cache:**
  - managed by hardware;  
transparent to the programmer
  - very strong influence on execution times



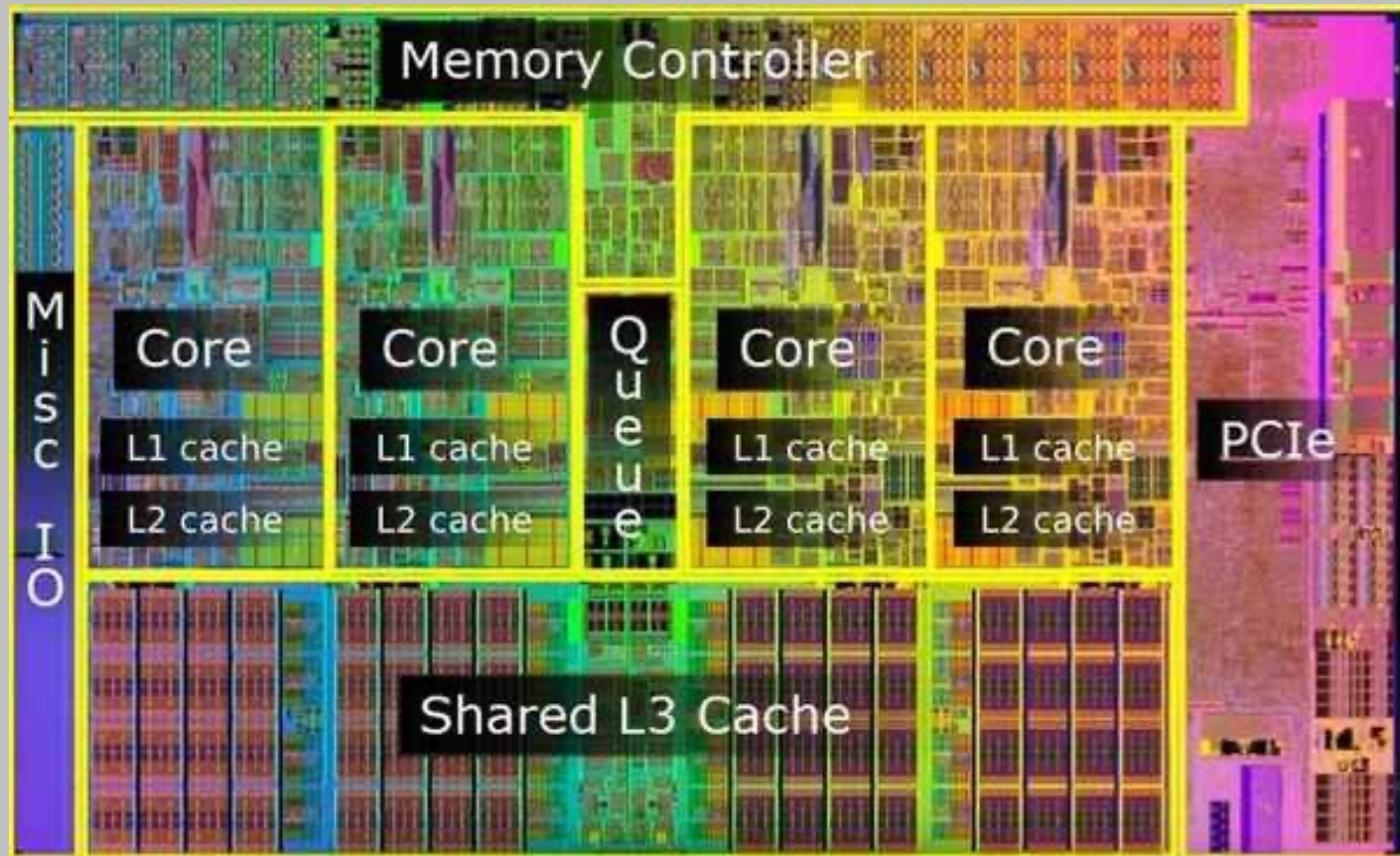
# Memory hierarchy: Examples

- Pentium-M „Dothan“ (single core), 2004
- 2 MB Level-2 Cache



# Memory hierarchy: Examples

- Intel Core i5-760 (quad core), 2010
- 8 MB Level-3 Cache





# Memory hierarchy: Examples

- Intel Core i7-5960X (octa core), 2014
- 20 MB Level-3 Cache

